TP de BiométrieSemestre 3

Benoît Simon-Bouhet

17 septembre 2024

Table des matières

Introduction

Objectifs

Ce livre contient l'ensemble du matériel (contenus, exemples, exercices…) nécessaire à la réalisation des travaux pratiques de **Biométrie** de l'EC '*Outils pour l'étude et la compréhension du vivant 2*' du semestre 3 de la licence Sciences de la Vie de La Rochelle Université.

Les 4 grands chapitres de ce livre correspondent aux 3 objectifs principaux de ces séances de TP et TEA :

1. **Vous faire découvrir les logiciels R et Rstudio** (chapitres Chapitre 1 et Chapitre 2) dans lesquels vous allez passer beaucoup de temps en licence puis en master. Si vous choisissez une spécialité de master qui implique de traiter des données (c'est-à-dire à peu près toutes les spécialités des Sciences de la Vie !) et/ou de communiquer des résultats d'analyses statistiques, alors `R` et `RStudio` devraient être les logiciels vers lesquels vous vous tournerez naturellement.

2. **Vous apprendre à faire des graphiques de qualités** dans `RStudio` et **vous faire prendre conscience de l'importance des visualisations graphiques** (chapitre Chapitre 3 : attention, ce chapitre est très long, ne vous laissez pas surprendre !) :

- d'une part, pour explorer des données inconnues et vous faire une première idée des informations qu'elles contiennent,
- d'autre part, pour vous permettre de formuler des hypothèses pertinentes et intéressantes concernant les systèmes que vous étudiez,
- et enfin, pour communiquer efficacement vos trouvailles à un public qui ne connaît pas vos données aussi bien que vous (cela inclut évidemment vos enseignants à l'issue de vos stages).

3. **Vous apprendre à manipuler efficacement des tableaux de données de grande taille** (chapitre Chapitre 4). Cela signifie que vous devriez être en mesure de sélectionner des variables (colonnes) d'un tableau, d'en créer de nouvelles en modifiant et/ou combinant des variables existantes, de filtrer des lignes spécifiques, etc.

À l'issue de ces TP et TEA, vous devriez donc être suffisamment à l'aise avec le logiciel `RStudio` pour y importer des données issues de tableurs, les manipuler pour les mettre dans un format permettant les représentations graphiques, et pour produire des graphiques pertinents, adaptés aux données dont vous disposez, et d'une qualité vous permettant de les intégrer sans honte à vos compte-rendus de TP et rapports de stages.

D'ailleurs, les données que vous serez amenés à traiter lors de vos stages, ou plus tard, lorsque vous serez en poste, ont souvent été acquises à grands frais, et au prix d'efforts importants. Il est donc de votre responsabilité d'en tirer le maximum. Et ça commence toujours (ou presque), par la manipulation de données dans `RStudio` et la réalisation de visualisations graphiques parlantes.

> **!** Important
>
> À partir de maintenant, tous les compte-rendus de TP que vous aurez à produire dans le cadre de la licence SV devront respecter les bonnes pratiques décrites dans ce document. En particulier, et sauf consigne contraire, les collègues de l'équipe pédagogique attendent que les graphiques que vous intégrerez à vos compte-rendus de TP soient systématiquement produits dans `RStudio`.

Organisation

Volume de travail

Les travaux pratiques et TEA de biométrie auront lieu entre le 9 septembre et le 11 octobre 2024. Chaque groupe aura 4 séances de TP de 90 minutes et 3 séances de TEA de 90 minutes selon le calendrier suivant :

Groupe 1

1. 17 sept, 13:15 - 14:45, MSI217
2. 23 sept, 18:30 - 20:00, MSI217
3. 3 oct, 09:45 - 11:15, PCM-INFO TRANS2
4. 8 oct, 09:45 - 11:15, PCM-SC5

Groupe 2

1. 17 sept, 18:30 - 20:00, MSI217
2. 23 sept, 16:45 - 18:15, MSI217
3. 30 sept, 18:30 - 20:00, MSI217
4. 8 oct, 08:00 - 09:30, PCM-SC5

Groupe 3

1. 9 sept, 18:30 - 20:00, MSI217
2. 18 sept, 18:30 - 20:00, MSI217
3. 2 oct, 18:30 - 20:00, MSI217
4. 7 oct, 08:00 - 09:30, PCM-SC6

Groupe 4

1. 9 sept, 16:45 - 18:15, MSI217
2. 17 sept, 09:45 - 11:15, MSI217
3. 1 oct, 18:30 - 20:00, MSI217
4. 8 oct, 16:45 - 18:15, MSI217

Groupe 5

1. 17 sept, 16:45 - 18:15, MSI217
2. 24 sept, 18:30 - 20:00, MSI217
3. 3 oct, 11:30 - 13:00, PCM-INFO TRANS2
4. 9 oct, 15:00 - 16:30, PCM-INFO TRANS2

Groupe 6

1. 18 sept, 16:45 - 18:15, MSI217

2. 24 sept, 16:45 - 18:15, MSI217
3. 30 sept, 09:45 - 11:15, MSI217
4. 11 oct, 08:00 - 09:30, MSI217

Groupe 7

1. 16 sept, 16:45 - 18:15, MSI217
2. 25 sept, 18:30 - 20:00, MSI217
3. 1 oct, 16:45 - 18:15, MSI217
4. 7 oct, 09:45 - 11:15, PCM-SC6

Groupe 8

1. 10 sept, 16:45 - 18:15, MSI217
2. 25 sept, 16:45 - 18:15, MSI217
3. 30 sept, 16:45 - 18:15, MSI217
4. 7 oct, 16:45 - 18:15, MSI217

Les TEA, ont tous lieu en distanciel synchrone, pour toute la promotion à la fois :

1. 21 sept, 09:45 - 11:15
2. 5 oct, 09:45 - 11:15
3. 7 oct, 18:30 - 20:00

Au total, chaque groupe aura donc 10h30 de TP et TEA. C'est très peu pour atteindre les objectifs fixés et il y aura donc évidemment du travail personnel à fournir en dehors de ces séances. **Pour chaque TP ou TEA** prévu à l'emploi du temps, j'estime qu'**une à deux heures de travail personnel est nécessaire** (soit 10 à 20 heures de travail personnel selon votre degré d'aisance, en plus des 10h30 prévues à votre emploi du temps au total). Attention donc : pensez bien à prévoir du temps dans vos plannings **en dehors des heurs prévues dans vos EDT** car le travail personnel est essentiel pour progresser dans cette matière. J'insiste sur l'importance de faire l'effort dès maintenant : vous allez en effet avoir des enseignements qui reposent sur l'utilisation de ces logiciels à chaque semestre de la licence, du S3 au S6. C'est maintenant qu'il faut acquérir des automatismes.

> ⚠ Travaillez régulièrement !
>
> J'insiste sur l'importance de travailler cette matière **régulièrement**. Vous devez vous y mettre dès maintenant

et y consacrer quelques heures chaque semaine. Interrogez vos collègues de L3 qui ont eu cet enseignement l'an dernier : il y a beaucoup de temps à y passer et il est hélas facile de prendre et d'accumuler du retard…

Modalités d'enseignement

Pour suivre cet enseignement vous pourrez utiliser les ordinateurs de l'université, mais je ne peux que vous encourager à utiliser vos propres ordinateurs, sous Windows, Linux ou MacOS. Lors de vos futurs stages et pour rédiger vos comptes-rendus de TP, vous utiliserez le plus souvent vos propres ordinateurs, autant prendre dès maintenant de bonnes habitudes en installant les logiciels dont vous aurez besoin tout au long de votre licence. Si vous ne possédez pas d'ordinateur, manifestez vous rapidement auprès de moi car des solutions existent (prêt par l'université, travail sur tablette via Posit cloud…).

> **!** Important
>
> L'essentiel du contenu de cet enseignement peut être abordé en autonomie, à distance, grâce à ce livre en ligne, aux ressources mises à disposition sur Moodle et à votre ordinateur personnel. Cela signifie que **la présence physique lors de ces séances de TP n'est pas obligatoire**.

Plus que des séances de TP classiques, considérez plutôt qu'il s'agit de **permanences non-obligatoires** : si vous pensez avoir besoin d'aide, si vous avez des points de blocage ou des questions sur le contenu de ce document ou sur les exercices demandés, alors venez poser vos questions lors des séances de TP. Vous ne serez d'ailleurs pas tenus de rester pendant 1h30 : si vous obtenez une réponse en 10 minutes et que vous préférez travailler ailleurs, vous serez libres de repartir !

De même, si vous n'avez pas de difficulté de compréhension et que vous n'avez pas de problème avec les exercices de ce livre en ligne, votre présence n'est pas requise. Si vous souhaitez malgré tout venir en salle de TP, pas de problème, vous y serez toujours les bienvenus. D'ailleurs, vous pouvez

aussi venir aux séances de permanences des autres groupes de TP (à supposer que vous soyez disponibles) si vous avez une question pressante ou si vous estimez que vous commencez à prendre du retard.

> **!** Important
>
> Attention, c'est bien la présence en salle de TP qui n'est pas obligatoire. **Faire le travail demandé dans le temps imparti reste bel et bien obligatoire !** Que vous fassiez le choix de venir ou non aux séances de TP, vous devez obligatoirement travailler cette matière lors des créneaux de TP et de TEA prévus dans votre emploi du temps ! Si vous considérez que ça n'est pas important puisque votre présence en salle de TP n'est pas requise, ou que vous avez plus urgent à faire dans l'immédiat, la chute sera brutale et le réveil douloureux…

Ce fonctionnement très souple a de nombreux avantages :

- vous disposez de plus de liberté et d'autonomie et vous avancez à votre rythme
- vous ne venez que lorsque vous en avez vraiment besoin
- celles et ceux qui se déplacent reçoivent une aide personnalisée "sur mesure"
- celles et ceux qui ne se déplacent pas reçoivent une aide à distance (voir plus bas)
- vous travaillez sur vos ordinateurs
- les effectifs étant réduits en salle, les conditions de travail sont idéales

Toutefois, pour que cette organisation fonctionne, cela demande de la rigueur de votre part, en particulier sur la régularité du travail que vous devez fournir. J'insiste sur le fait que si la présence en salle de TP n'est pas requise, le travail demandé est bel et bien obligatoire ! Si vous venez en salle de TP sans avoir travaillé en amont, sans avoir expérimenté et sans avoir de question à poser, ces séances ne seront pas plus utiles qu'un TP classique où vous passerez votre séance à lire ce livre en ligne. Vous perdrez donc en partie votre temps. De même, si vous attendez la 4e séance pour vous y mettre sérieusement, vous irez droit dans le mur. Je le répète, outre les 10h30 de TP/TEA prévus dans vos emplois du temps,

vous devez prévoir entre 10 et 20 heures de travail personnel supplémentaire.

Je vous laisse donc une grande liberté d'organisation. À vous d'en tirer le maximum et de faire preuve de la maturité et du sérieux nécessaires. Le rythme auquel vous devriez avancer est présenté séance par séance, dans la section intitulée "Progression conseillée" un peu plus bas.

Utilisation de Slack

Outre les séances de permanences non-obligatoires, nous échangerons aussi sur l'application Slack, qui fonctionne un peu comme un chat privé. Slack facilite la communication des équipes et permet de travailler ensemble. Créez-vous un compte en ligne et installez le logiciel sur votre ordinateur dès maintenant (il existe aussi des versions pour tablettes et smartphones). Lorsque vous aurez installé le logiciel, revenez ici et cliquez sur ce lien pour vous connecter à notre espace de travail commun intitulé `L2 SV 24-25 / EC outils` (ce lien expire au bout d'un mois, donc suivez-le dès aujourd'hui et faites moi signe s'il n'est plus valide).

Vous verrez que 2 "chaînes" sont disponibles :

- `#général` : c'est là que les questions liées à l'organisation générale du cours, des TP, des TEA et des évaluations doivent être posées. C'est également ici que je posterai les annonces importantes concernant l'organisation des séances et de l'évaluation.
- `#questions-rstudio` : c'est ici que toutes les questions pratiques liées à l'utilisation de `R` et `RStudio` devront êtres posées. Problèmes de syntaxe, problèmes liés à l'interface, à l'installation des packages ou à l'utilisation des fonctions, à la création des graphiques, à la manipulation des tableaux… Tout ce qui concerne directement les logiciels sera traité ici. Vous êtes libres de poser des questions, de poster des captures d'écran, des morceaux de code, des messages d'erreur. Et **vous êtes bien entendus vivement encouragés à vous entraider et à répondre aux questions de vos collègues.** Je n'interviendrai ici que pour répondre aux questions laissées sans réponse ou si les réponses apportées sont inexactes. Le fonctionnement est celui d'un

> forum de discussion instantané. Vous en tirerez le plus grand bénéfice en participant et en n'ayant pas peur de poser des questions, même si elles vous paraissent idiotes. Rappelez-vous toujours que si vous vous posez une question, d'autres se la posent aussi probablement.

Ainsi, quand vous travaillerez à vos TP ou TEA, que vous soyez installés chez vous ou en salle de TP, prenez l'habitude de garder Slack ouvert sur votre ordinateur. Même si vous n'avez pas de question à poser, votre participation active pour répondre à vos collègues est souhaitable et souhaitée. Je vous incite donc fortement à vous **entraider** : c'est très formateur pour celui qui explique, et celui qui rencontre une difficulté a plus de chances de comprendre si c'est quelqu'un d'autre qui lui explique plutôt que la personne qui a rédigé les instructions mal comprises.

Ce document est fait pour vous permettre d'avancer en autonomie et vous ne devriez normalement pas avoir beaucoup besoin de moi si votre lecture est attentive. L'expérience montre en effet que la plupart du temps, il suffit de lire correctement les paragraphes précédents et/ou suivants pour obtenir la réponse à ses questions. J'essaie néanmoins de rester disponible sur Slack pendant les séances de TP et de TEA de tous les groupes. Cela veut donc dire que même si votre groupe n'est pas en TP, vos questions ont des chances d'être lues et de recevoir des réponses dès que d'autres groupes sont en TP ou TEA. Vous êtes d'ailleurs encouragés à échanger sur Slack aussi pendant vos phases de travail personnels.

Progression conseillée

Pour apprendre à utiliser des logiciels comme `R` et `RStudio`, il faut faire les choses soi-même, ne pas avoir peur des messages d'erreurs (il faut d'ailleurs apprendre à les déchiffrer pour comprendre d'où viennent les problèmes), essayer maintes fois, se tromper beaucoup, recommencer, et surtout, ne pas se décourager. J'utilise ce logiciel presque quotidiennement depuis bientôt 20 ans et à chaque session de travail, je rencontre des messages d'erreur. Avec suffisamment d'habitude, on apprend à les déchiffrer, et on corrige les problèmes en quelques secondes. Ce livre est conçu pour vous faciliter la

tâche, mais ne vous y trompez pas, vous rencontrerez des difficultés, et c'est normal. C'est le prix à payer pour profiter de la puissance du meilleur logiciel permettant d'analyser des données, de produire des graphiques de qualité et de réaliser toutes les statistiques dont vous aurez besoin d'ici la fin de vos études et au-delà.

Pour que cet apprentissage soit le moins problématique possible, il convient de prendre les choses dans l'ordre. C'est la raison pour laquelle les chapitres de ce livre doivent être lus attentivement et **dans l'ordre**, et les exercices d'application faits au fur et à mesure de la lecture.

Idéalement, voilà les étapes que vous devriez avoir franchies à chaque séance :

1. La première séance est consacrée à l'installation des logiciels, à la découverte de l'environnement de travail, des `RProjects`, des packages et des scripts. Avant votre deuxième séance, vous devrez avoir terminé le premier chapitre de ce livre, vous devrez être capables de créer un `Rproject` et un script, de télécharger et d'installer des packages, et d'exécuter des commandes simples dans votre script. **Attention**, dans la section Section 1.3.4.1, **vous devrez suivre plusieurs tutoriels sur la plateforme `DataCamp`** (ça prend du temps !) et vous devrez connaitre les types d'objets suivants : vecteurs, facteurs, data.frames (et tibbles). Suivez bien les indications de cette section : seuls les tutoriels qui concernent ces objets sont à suivre, et pas ceux qui concernent les matrices ni les listes. Après chaque tutoriel, revenez à ce livre en ligne et cliquez sur le lien du tutoriel suivant. De cette façon, vous accéderez aux versions françaises des tutoriels, et la plateforme ne vous demandera pas de prendre une inscription payante.

2. La deuxième séance est consacrée à la découverte des premiers jeux de données et du package `ggplot2`. Avant votre troisième séance, vous devrez avoir terminé la section Section 3.6.2. À ce stade, vous devriez donc avoir compris comment charger en mémoire les jeux de données disponibles dans un packages, vous devriez connaitre la syntaxe de base permettant de faire toutes sortes de graphiques avec `ggplot2` avec **une unique**

variable numérique ou catégorielle. À terme, vous devrez être capables de choisir des graphiques appropriés selon le nombre et la nature des variables dont vous disposez, mais à ce stade, on ne demande rien de complexe. Vous devriez toutefois être capable de faire des barplots et des histogrammes, et vous devriez être capable de distinguer ces deux types de représentations graphiques : dans quelle situation utilise-t-on plutôt l'un ou l'autre ? Quel type de variable utiliser pour produire une histogramme ou un diagramme bâtons ? À quoi correspond l'axe des ordonnées de ces graphiques et comment est-il obtenu ?

3. La troisième séance est consacrée à la découverte de graphiques permettant de visualiser les **relations entre plusieurs variables** : 2, 3 ou plus. Vous verrez également comment améliorer la qualité de vos graphiques en vue de leur intégration dans un compte-rendu, un rapport ou une présentation. Avant votre dernière séance de TP, vous devrez avoir terminé le chapitre Chapitre 3. Vous devriez donc être capable de faire, outre les graphiques de la semaine précédente, des nuages de points, des stripcharts, des graphiques en lignes et des boîtes à moustaches (ou boxplots). Vous devriez également être capable de faire des sous-graphiques par catégories (`facet`), de choisir un thème et des palettes de couleurs appropriées, et de légender/annoter correctement vos graphiques. Attention, ce chapitre est (très) long ! Travaillez régulièrement et ne vous découragez pas !

4. La quatrième et dernière séance est consacrée à la manipulation des tableaux de données. À l'issue de cette séance, vous devrez avoir atteint la fin du document, et vous devrez donc être capable de sélectionner des colonnes dans un tableau, de filtrer des lignes, et de créer de nouvelles variables. Vous devrez être capables d'enchaîner correctement les étapes suivantes : ouvrir le logiciel > créer un `Rproject` > créer un script > mettre en mémoire les packages utiles > importer des données > mettre en forme ces données > faire un ou des graphiques informatifs et correctement mis en forme.

> ⚠ Attention au retard
>
> Relisez régulièrement cette section (chaque semaine !) pour savoir si les objectifs fixés ici sont atteints et vous assurez que vous ne prenez pas de retard…

Évaluation(s)

L'évaluation de la biométrie du semestre 3 sera assurée par Benoît Lebreton. Vous êtes susceptibles d'être interrogés sur tout ce qui est décrit dans ce livre en ligne.

Les exercices de ce livre en ligne ne seront ni ramassés, ni notés : ils sont proposés pour que vous puissiez mettre en application les notions récemment apprises et afin d'évaluer votre propre progression et vos apprentissages. Ils doivent vous aider à mettre le doigt sur les choses que vous n'avez pas comprises afin que vous puissiez me poser des questions sur Slack ou lors des séances de permanences. Tout ce que nous voyons en TP et TEA devra être acquis à la fin des TP. Les TP de biométrie du semestre 4 (et des 2 semestres de L3) supposent en effet que les notions abordées ici soient parfaitement acquises. En particulier, les étapes décrites précédemment doivent être maîtrisées (ouvrir le logiciel > créer un projet > créer un script > mettre en mémoire les packages utiles > importer des données > mettre en forme ces données > faire un ou des graphiques informatifs et correctement mis en forme).

La grille critériée suivante sera utilisée pour l'évaluation :

Critères évalués	Acquis	En voie d'acquisition	Non acquis
Connaître le vocabulaire et savoir utiliser la terminologie associée à la biométrie	Les phrases sont rédigées avec un vocabulaire précis et exact.	Les phrases sont rédigées avec des imprécisions mais le sens de la phrase reste compréhensible.	Les phrases ne sont pas compréhensibles ou fausses. Absence de rédaction
Connaître les différents types de variables et leurs échelles de variations, et les limites de l'utilisation des différentes variables.	Les variables utilisées et leurs échelles de variations sont correctement définies et utilisées.	Des erreurs persistent dans l'utilisation des types de variables et de leurx échelles, mais ces erreurs restent mineures et n'affectent pas les résultats et les conclusions obtenues lors de l'analyse des données.	Les variables et les échelles sont fausses ou non définies. Cela impacte les conclusions obtenue suite à l'analyse des données ou rend leur analyses impossible.
Savoir construire un tableau et un diagramme de distribution de fréquences	Le tableau et/ou le diagramme sont justes (i.e. les données sont correctement décrites), clairs (i.e. classes correctement définies, échelles correctes) et complets (i.e. légendes, titres, etc… sont présents).	Le tableau et/ou le diagramme présentent des inexactitudes ou des éléments manquent, mais l'interprétation correcte des données peut quand même être fait sans que cela impacte les conclusions.	Le tableau et/ou le diagramme présentent des erreurs qui ne permettent pas d'interpréter correctement les données, ce qui peut amener à des conclusions inexactes.
Savoir utliser des indices de statistiques descriptives pertinents compte tenu de la nature des variables disponibles et de la question posée afin de synthétiser l'information collectée	Des indices de statistiques descriptives de position et de dispersion sont fournis, pour chaque variable utile, et à la bonne échelle d'observation. Les indices utilisés sont adaptés au type de variable, à l'échelle et à la question à laquelle on souhaite répondre.	Des indices de statistiques descriptives sont fournis, mais ils ne permettent pas de décrire complètement le jeu de données. Les indices utlisés manquent de pertinence sans pour autant que cela impacte les conclusions qui pourront être faites suite à l'analyse de ces indices.	Aucun indice de statistiques descriptive n'est fourni, ou si des indices sont fournis, ils sont mal choisis ou calculés. Les indices fournis ne permettent pas d'accéder à des informations pertinentes vis-à-vis des questions posées
Savoir représenter des données sous forme d'un graphique adapté, en tenant compte de la nature des variables et de la question posée.	Les graphiques produits permettent au lecteur d'interpréter les données et de répondre à la question posée. Les graphiques sont adaptés à la nature des variables et aux questions posées, et leur mise en forme est correcte (p. ex. catégories faciles à distinguer, axes correctement légendés, unités appropriées...). Quelques oublis ou erreurs minimes peuvent subsister (fautes d'orthographe, symboles trop ressemblants, code peu clair), mais ils gênent peu la lecture et l'interprétation des graphiques produits.	Des graphiques sont produits, ils permettent d'interpréter les données et de répondre à la question posée, mais ils ne sont pas idéaux : soit leur mise en forme n'est pas à la hauteur (mauvais choix de couleurs, de symboles, axes non ou incorrectement légendés, catégories impossibles à distinguer...), soit la nature des graphiques produits n'est pas idéale (diagrammes bâtons, histogrammes, nuages de points, boîtes à moustaches etc.)	Les graphiques fournis ne sont pas adaptés à la nature des variables disponibles ou à la questions posée, ou la mise en forme inadaptée rend leur lecture et leur bonne interprétation impossible

Licence

lien vers la licence Creative Commons et indiquer si des modifications du contenu original ont été effectuées. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'auteur vous soutient ou soutient la façon dont vous avez utilisé son travail.

Pas d'Utilisation Commerciale : vous n'êtes pas autorisé à faire un usage commercial de cet ouvrage, ni de tout ou partie du matériel le composant. Cela comprend évidemment la diffusion sur des plateformes de partage telles que studocu.com qui tirent profit d'œuvres dont elles ne sont pas propriétaires, souvent à l'insu des auteurs.

Pas de modifications : dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'ouvrage original, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'ouvrage modifié.

Pas de restrictions complémentaires : vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser cet ouvrage dans les conditions décrites par la licence.

1 R et RStudio : les bases

1.1 Préambule

Avant de commencer à explorer des données dans `R`, il y a plusieurs concepts clés qu'il faut comprendre en premier lieu :

1. Que sont `R` et `RStudio` ?
2. Comment s'y prend-on pour coder dans `R` ?
3. Que sont les `packages` ?

Une bonne maîtrise des éléments présentés dans ce chapitre est indispensable pour aborder sereinement les chapitres suivants, à commencer par le chapitre Chapitre 2, qui présente un jeu de données que nous explorerons en détail un peu plus tard. Lisez donc attentivement ce chapitre et faites bien tous les exercices demandés.

Ce chapitre est en grande partie basé sur les 3 ressources suivantes que je vous encourage à consulter si vous souhaitez obtenir plus de détails :

1. L'ouvrage intitulé ModernDive, de Chester Ismay et Albert Y. Kim. Une bonne partie de ce livre est très largement inspirée de cet ouvrage. C'est en anglais, mais c'est un très bon texte d'introduction aux statistiques sous `R` et `RStudio`.
2. L'ouvrage intitulé Getting used to R, RStudio, and R Markdown de Chester Ismay, comprend des podcasts (en anglais toujours) que vous pouvez suivre en apprenant.
3. Les tutoriels en ligne de DataCamp. DataCamp est une plateforme de e-learning accessible depuis n'importe quel navigateur internet et dont la priorité est l'enseignement des "data sciences". Leurs tutoriels vous aideront à apprendre certains des concepts de développés dans ce livre.

> **Important**
>
> Avant d'aller plus loin, rendez-vous sur le site de DataCamp et créez-vous un compte gratuit.

1.2 Que sont `R` et `RStudio` ?

Pour l'ensemble de ces TP, j'attends de vous que vous utilisiez `R` *via* `RStudio`. Les utilisateurs novices confondent souvent les deux. Pour tenter une analogie simple :

- `R` est le moteur d'une voiture
- `RStudio` est l'habitacle, le tableau de bord, les pédales…

Si vous n'avez pas de moteur, vous n'irez nulle part. En revanche, un moteur sans tableau de bord est difficile à manœuvrer. Il est en effet beaucoup plus simple de faire avancer une voiture depuis l'habitacle, plutôt qu'en actionnant à la main les câbles et leviers du moteur.

En l'occurrence, `R` est un langage de programmation capable de produire des graphiques et de réaliser des analyses statistiques, des plus simples aux plus complexes. `RStudio` est un "emballage" qui rend l'utilisation de `R` plus aisée. `RStudio` est ce qu'on appelle un IDE ou "Integrated Development Environment". On peut utiliser `R` sans `RStudio`, mais c'est nettement plus compliqué, nettement moins pratique.

1.2.1 Installation

> **Avertissement**
>
> Si vous travaillez exclusivement sur les ordinateurs de l'Université, vous pouvez passer cette section. En revanche, si vous souhaitez utiliser `R` et `RStudio` sur votre ordinateur personnel, alors lisez attentivement la suite !

Avant tout, vous devez télécharger et installer `R`, **puis** `RStudio`, dans cet ordre :

1. Téléchargez et installez `R`

- Vous devez installer ce logiciel en premier.
- Cliquez sur le lien de téléchargement qui correspond à votre système d'exploitation, puis, sur "base", si vous êtes sous Windows, sur "`R-4.4.1.pkg`" si vous êtes sous Mac avec processeur Intel, ou sur `R-4.4.1-arm64.pkg` si vous êtes sous Mac avec processeur M1 ou M2 (sous Mac, cliquez sur le Menu , puis sur "À propos de ce Mac" et regardez à la rubrique "Processeur"), et suivez les instructions.

2. Téléchargez et installez `RStudio`

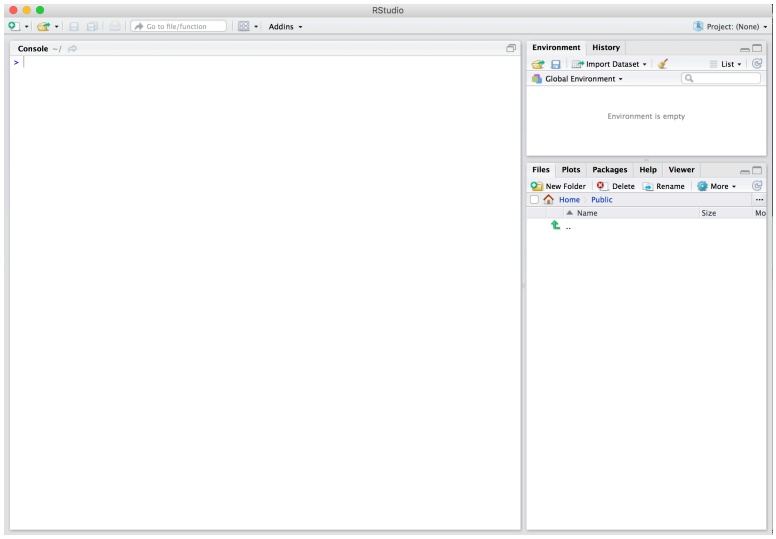
- Cliquez sur "RStudio Desktop", puis sur "Download RStudio Desktop".
- Choisissez la version gratuite et cliquez sur le lien de téléchargement qui correspond à votre système d'exploitation.

1.2.2 Utiliser `R` depuis `RStudio`

Puisqu'il est beaucoup plus facile d'utiliser `Rstudio` pour interagir avec `R`, nous utiliserons exclusivement l'interface de `RStudio`. Après les installations réalisées à la Section 1.2.1, vous disposez de 2 nouveaux logiciels sur votre ordinateur. `RStudio` ne peut fonctionner sans `R`, mais nous travaillerons exclusivement dans `RStudio` :

- `R`, ne pas ouvrir ceci : 
- `RStudio`, ouvrir cela : 

À l'université, vous trouverez `RStudio` dans le menu Windows. Quand vous ouvrez RStudio pour la première fois, vous devriez obtenir une fenêtre qui ressemble à ceci :



Prenez le temps d'explorer cette interface, cliquez sur les différents onglets, ouvrez les menus, allez faire un tour dans les préférences du logiciel pour découvrir les différents panneaux de l'application, en particulier la Console dans laquelle nous exécuterons très bientôt du code `R`.

1.3 Comment exécuter du code `R` ?

Contrairement à d'autres logiciels comme Excel, STATA ou SAS qui fournissent des interfaces où tout se fait en cliquant avec sa souris, `R` est un langage interprété, ce qui signifie que vous devez taper des commandes, écrites en code `R`. C'est-à-dire que vous devez **programmer** en `R` (j'utilise les termes "coder" et "programmer" de manière interchangeable dans ce livre).

Il n'est pas nécessaire d'être un programmeur pour utiliser `R`, néanmoins, il est nécessaire de programmer ! Il existe en effet un ensemble de concepts de programmation de base que les utilisateurs `R` doivent comprendre et maîtriser. Par conséquent, bien que ce livre ne soit pas un livre sur la programmation, vous en apprendrez juste assez sur ces concepts de programmation de base pour explorer et analyser efficacement des données.

1.3.1 La console

La façon la plus simple d'interagir avec `RStudio` (mais pas du tout la meilleure !) consiste à taper directement des commandes que `R` pourra comprendre **dans la Console**.

Cliquez dans la console (après le symbole `>`) et tapez ceci, sans oublier de valider en tapant sur la touche `Entrée` :

```
3 + 8
```

```
[1] 11
```

Félicitations, vous venez de taper votre première instruction `R` : vous savez maintenant faire des additions !

Dans la version en ligne de ce livre (en html), à chaque fois que du code `R` sera fourni, il apparaîtra dans un cadre grisé avec une ligne bleue à gauche, comme ci-dessus. Vous pourrez toujours taper dans `RStudio`, les commandes qui figurent dans ces blocs de code, afin d'obtenir vous même les résultats souhaités. Dans ce livre, lorsque les commandes `R` produisent des résultats, ils sont affichés juste en dessous des blocs de code. Enfin, en passant la souris sur les blocs de code, vous verrez apparaître, à droite, une icône de presse-papier qui vous permettra de copier-coller les commandes du livre dans la console de `RStudio` ou, très bientôt, dans vos scripts.

> ⚠ Les risques du "copier-coller" 📋
>
> Attention : il est fortement conseillé de réserver les copier-coller aux blocs de commandes de (très) grande taille, ou en cas d'erreur de syntaxe inexplicable. L'expérience a en effet montré qu'**on apprend beaucoup mieux en tapant soi-même les commandes**. Ça n'est que comme cela que l'on peut prendre conscience de toutes les subtilités du langage (par exemple, faut-il mettre une virgule ou un point, une parenthèse ou un crochet, le symbole moins ou un tilde, etc.). Je vous conseille donc de taper vous-même les commandes autant que possible.

1.3.2 Les scripts

Taper du code directement dans la console est probablement la pire façon de travailler dans `RStudio`. Cela est parfois utile pour faire un rapide calcul, ou pour vérifier qu'une commande fonctionne correctement. Mais la plupart du temps, **vous devriez taper vos commandes dans un script**.

> ❗ Définition importante !
>
> Un script est un fichier au format "texte brut" (cela signifie qu'il n'y a pas de mise en forme et que ce fichier peut-être ouvert par n'importe quel éditeur de texte, y compris les plus simples comme le bloc notes de Windows), dans lequel vous pouvez taper :
>
> 1. des instructions qui seront comprises par `R` comme si vous les tapiez directement dans la console
> 2. des lignes de commentaires, qui doivent obligatoirement commencer par le symbole `#`.

Les avantages de travailler dans un script sont nombreux :

1. Vous pouvez sauvegarder votre script à tout moment (vous devriez prendre l'habitude de le sauvegarder très régulièrement). Vous gardez ainsi la trace de toutes les commandes que vous avez tapées.
2. Vous pouvez aisément partager votre script pour collaborer avec vos collègues de promo et enseignants.
3. Vous pouvez documenter votre démarche et les différentes étapes de vos analyses. **Vous devez ajouter autant de commentaires que possible**. Cela permettra à vos collaborateurs de comprendre ce que vous avez fait. Et dans 6 mois, cela vous permettra de comprendre ce que vous avez fait. Si votre démarche vous paraît cohérente aujourd'hui, il n'est en effet pas garanti que vous vous souviendrez de chaque détail quand vous vous re-plongerez dans vos analyses dans quelques temps. Donc aidez-vous vous même en commentant vos scripts dès maintenant.
4. Un script bien structuré, bien indenté (avec les bons retours à la ligne, des sauts de lignes, des espaces, bref, de l'air) et clair permet de rendre vos analyses répétables.

> Si vous passez 15 heures à analyser un tableau de données précis, il vous suffira de quelques secondes pour analyser un nouveau jeu de données similaire : vous n'aurez que quelques lignes à modifier dans votre script original pour l'appliquer à de nouvelles données.

Vous pouvez créer un script en cliquant dans le menu "File > New File > R Script". Un nouveau panneau s'ouvre dans l'application. Pensez à sauvegarder immédiatement votre nouveau script en cliquant dans le menu "File > Save" ou "File > Save as…". Il faut pour cela lui donner un nom et choisir un emplacement sur votre disque dur.

! Où sauvegarder vos scripts ?

Je vous encourage vivement à créer, sur votre disque dur, un nouveau dossier spécifique, que vous nommerez par exemple `BiometrieS3`. Il est important que le nom de ce dossier ne contienne pas de caractères spéciaux (*e.g.* accents, cédilles, apostrophes, espaces, etc.). Ce dossier devrait être facilement accessible : vous y enregistrerez tous vos scripts, vos jeux de données, vos graphiques, etc.
Si vous travaillez sur les ordinateurs de l'université, créez obligatoirement votre dossier sur le disque `W:\`. Il s'agit de votre espace personnel sur le réseau de l'université. Cela vous garantit que vous retrouverez votre script la prochaine fois, même si vous utilisez un ordinateur différent.

À partir de maintenant, vous ne devriez plus taper de commande directement dans la console. Tapez systématiquement vos commandes dans un script et sauvegardez-le régulièrement.

Pour exécuter les commandes du script dans la console, il suffit de placer le curseur sur la ligne contenant la commande et de presser les touches `ctrl + enter` (ou `command + enter` sous macOS). Si un message d'erreur s'affiche dans la console, c'est que votre instruction était erronée. Modifiez la directement dans votre script et pressez à nouveau les touches `ctrl + enter` (ou `command + enter` sous macOS) pour tenter à nouveau votre chance. Idéalement, votre script ne devrait contenir que des commandes qui fonctionnent et des

commentaires expliquant à quoi servent ces commandes.

Voici un exemple de script que je ne vous demande pas de reproduire. Lisez simplement attentivement son contenu :

```
# Penser à installer le package ggplot2 si besoin
# install.packages("ggplot2")

# Chargement du package
library(ggplot2)

# Mise en mémoire des données de qualité de l'air à New-York de mai à
# septembre 1973
data(airquality)

# Affichage des premières lignes du tableau de données
head(airquality)

# Quelle est la structure de ce tableau ?
str(airquality)

# Réalisation d'un graphique présentant la relation entre la concentration
# en ozone atmosphérique en ppb et la température en degrés Farenheit
ggplot(data = airquality, mapping = aes(x = Temp, y = Ozone)) +
  geom_point() +
  geom_smooth(method = "loess")

# On constate une augmentation importante de la concentration d'ozone
# pour des températures supérieures à 75ºF
```

Même si vous ne comprenez pas encore les commandes qui figurent dans ce script (ça viendra !), voici ce que vous devez en retenir :

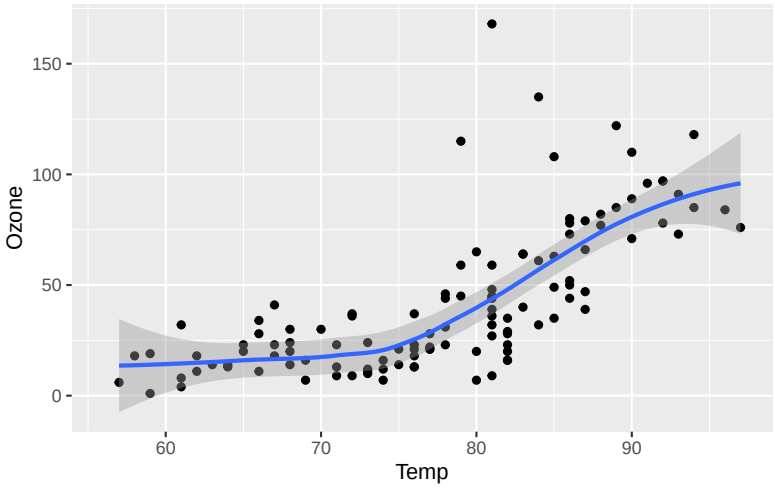
1. Le script contient plus de lignes de commentaires que de commandes `R`.
2. Chaque étape de l'analyse est décrite en détail.
3. Les 2 dernières lignes du script décrivent les résultats obtenus (ici, un graphique).
4. Seules des commandes pertinentes et qui fonctionnent ont été conservées dans ce script.
5. Chaque ligne de commentaire commence par `#`. Il est ainsi possible de conserver certaines commandes `R` dans le script, "pour mémoire", sans pour autant

qu'elle ne soient exécutées. C'est le cas pour la ligne `# install.packages("ggplot2")`.

Si j'exécute ce script dans la console de `RStudio` (en sélectionnant toutes les lignes et en pressant les touches `ctrl + enter` ou `command + enter` sous macOS), voilà ce qui est produit :

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

```
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

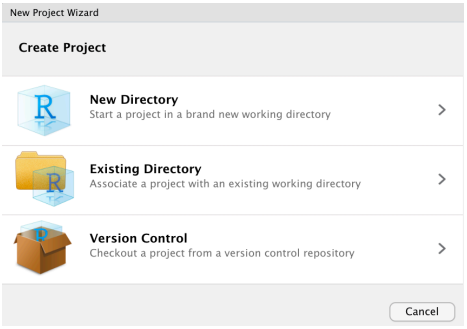


1.3.3 Les projets, ou `Rprojects`

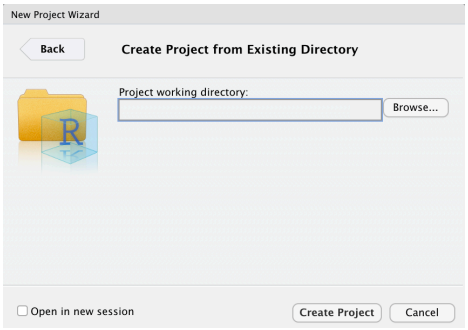
Pour travailler le plus efficacement possible avec `RStudio`, vous devriez créer, à l'intérieur de votre dossier de travail, un

nouveau fichier très particulier, qui s'appelle, dans le jargon de `RStudio`, un **`Rproject`**.

Pour le créer, cliquez simplement dans le Menu "File > New Project…". Cette boîte de dialogue devrait apparaître :



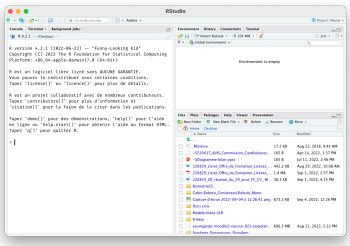
Choisissez "Existing Directory", puis, dans la boîte de dialogue suivante :



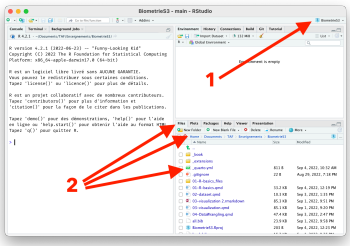
cliquez sur "Browse…", naviguez jusqu'au dossier que vous avez créé plus tôt sur votre disque dur et qui contient votre script, puis cliquez sur "Create Project". La fenêtre de `RStudio` se ferme, puis une nouvelle fenêtre vierge apparaît. En apparence, rien n'a changé ou presque. Pourtant :

1. en haut à droite de la fenêtre de `RStudio`, le logiciel indique maintenant que vous êtes bel et bien à l'intérieur d'un `Rproject`. Au lieu de `Project: (None)`, on lit maintenant le nom du `Rproject` (chez moi, `BiometrieS3`)

2. dans le quart inférieur droit de l'interface, l'onglet "Files" ne présente plus le même aspect. Avant de créer le `Rproject`, cet onglet présentait le chemin vers le dossier utilisé par défaut par le logiciel, ainsi que son contenu. Il s'agissait d'un dossier système auquel il vaut mieux ne pas toucher pour éviter les problèmes. Après la création du `Rproject`, l'onglet "Files" indique le contenu du dossier contenant le projet. Autrement dit, c'est ici que vous trouverez vos scripts, tableaux de données dans différents formats, figures sauvegardées, etc. Dans cet onglet, vous pouvez donc cliquer sur le nom de votre script pour l'ouvrir à nouveau, le modifier, l'exécuter…



(a) Sans `Rproject`



(a) Avec `RProject`

3. un nouveau fichier portant l'extension `.Rproj` a été créé dans votre dossier de travail. La prochaine fois que vous voudrez travailler dans `RStudio`, il vous suffira de double-cliquer sur ce fichier dans l'explorateur de fichier de Windows ou le Finder de MacOS, pour que `RStudio` s'ouvre, et que vous retrouviez tous vos fichiers et scripts de la fois précédente

BiometrieS3.Rproj

Pour vérifier que tout s'est bien passé jusqu'ici, tapez la commande suivante dans votre script puis envoyez-la dans la console en pressant les touches `ctrl + entrée` (ou `command + entrée` sous MacOS).

```
getwd()
```

`RStudio` doit vous afficher, dans la console, le chemin jusqu'à votre répertoire de travail ou "Working Directory" en anglais (`getwd()` est l'abréviation de "GET Working Directory"). Si tout s'est bien passé, ce chemin doit être celui du dossier qui contient votre script et le fichier `.Rproj` que vous venez de créer. Si ce n'est pas le cas, reprenez calmement toutes les étapes décrites depuis le début de la Section 1.3.2. Si ça ne fonctionne toujours pas, contactez-moi sur Slack.

! Pour résumer… 💻

Les `Rprojects` sont un moyen très pratique de travailler efficacement dans `RStudio` car ils permettent de gérer facilement la question du répertoire de travail. Lorsque vous envisagez de travailler sur un nouveau sujet/projet/jeu de données/compte-rendu de TP…, les étapes à suivre, pour vous mettre dans une configuration idéale qui vous évitera bien des problèmes par la suite, sont donc les suivantes :

1. Sur votre ordinateur, créez un nouveau dossier avec un nom simple, et à un endroit facile d'accès (pas de caractères spéciaux dans le chemin du dossier si possible)
2. Démarrez `RStudio`
3. Dans le logiciel, cliquez dans le menu "File > New Project…"
4. Choisissez "Existing Directory", puis naviguez jusqu'au dossier que vous venez de créer
5. Cliquez sur "Create Project"
6. Créer un nouveau script (menu "File > New File > R script")
7. Donnez un nom à votre script (menu "File > Save As…") pour le sauvegarder. Par défaut, `RStudio` vous propose d'enregistrer votre script dans le dossier de votre `Rproject`, ce qui est parfait.
8. Tapez `getwd()` dans votre script et exécutez cette commande en l'envoyant dans la console.

Si le chemin qui s'affiche est celui du dossier contenant votre `Rproject` et votre script, félicitation, vous êtes

prêt · e à travailler. Avec un peu d'habitude, ces étapes ne prennent qu'une à deux minutes.

1.3.4 Concepts de base en programmation et terminologie

Après ces considérations techniques sur l'utilisation et les réglages de `RStudio`, nous entrons maintenant dans le vif du sujet avec la découverte des premiers éléments de syntaxe du langage `R`.

1.3.4.1 Objets, types, vecteurs, facteurs et tableaux de données

Pour vous présenter les concepts de base et la terminologie de la programmation dont nous aurons besoin, vous allez suivre des tutoriels en ligne sur le site de DataCamp. Pour cette première prise en main, tout va maintenant se passer dans votre navigateur internet, et vous pouvez donc mettre de côté `RStudio` pour l'instant. Vous allez voir que l'interface de DataCamp ressemble à une version simplifiée de l'éditeur de script et de la console de `RStudio` : vous n'aurez pas à vous soucier des réglages, de `Rprojects` ou de sauvegarder quoi que ce soit. Si vous avez correctement créé votre compte gratuit DataCamp comme indiqué au tout début de la Chapitre 1, votre progression sera sauvegardée automatiquement. Il vous suffit de cliquer sur les liens direct ci-dessous pour démarrer les tutoriels en ligne.

Avant de démarrer, quelques précisions :

- pour chaque tutoriel que je vous demande de suivre, j'indique ci-dessous une liste des concepts de programmation qui sont couverts. N'hésitez pas à vous y référer (et à y revenir) tout au long du semestre si vous avez oublié certaines choses
- ce tutoriel DataCamp contient 6 chapitres. Seuls les chapitres 1, 2, 4, et 5 doivent être suivis. Nous ne travaillerons pas sur les matrices ni sur les listes
- à la fin de chaque chapitre du tutoriel, revenez à ce livre en ligne pour cliquer sur le lien direct ver le chapitre

suivant. Procéder ainsi vous évitera de suivre des chapitres inutiles du tutoriel, et cela vous permettra également d'éviter les demandes d'inscriptions payantes à DataCamp

Il est important de noter que, bien que ces tutoriels sont d'excellentes introductions, une lecture seule, même attentive, est insuffisante pour un apprentissage en profondeur et une rétention à long terme. Il faut pour cela **pratiquer** et **répéter**. Outre les exercices demandés dans DataCamp, que vous devez effectuer directement dans votre navigateur, je vous encourage à prendre des notes, à multiplier les essais, directement dans la console de `RStudio`, ou, de préférence, dans un script que vous annoterez, pour vous assurer que vous avez bien compris chaque partie.

Allez maintenant découvrir le cours d'introduction à R sur DataCamp, et cliquez sur les liens des chapitres ci-dessous. Au fur et à mesure de votre travail, notez les termes importants et ce à quoi ils font référence.

- Chapitre 1 : introduction
  - La console : l'endroit où vous tapez des commandes
  - Les objets : où les valeurs sont stockées, comment assigner des valeurs à des objets
  - Les types de données : entiers, doubles/numériques, caractères et logiques
- Chapitre 2 : vecteurs
  - Les vecteurs : des collections de valeurs du même type
- Chapitre 4 : les facteurs
  - Des données catégorielles (et non pas *numériques*) représentées dans `R` sous forme de `factor`s
- Chapitre 5 : les jeux de données ou `data.frame`
  - Les `data.frame`s sont similaires aux feuilles de calcul rectangulaires que l'on peut produire dans un tableur. Dans `R`, ce sont des objets rectangulaires (des tableaux !) contenant des jeux de données : les lignes correspondent aux observations et les colonnes aux variables décrivant les observations. La

plupart du temps, c'est le format de données que nous utiliserons. Plus de détails dans le Chapitre 2

Avant de passer à la suite, il nous reste 2 grandes notions à découvrir dans le domaine du code et de la syntaxe afin de pouvoir travailler efficacement dans `R` : les opérateurs de comparaison d'une part, et les fonctions d'autre part. Pour les découvrir et expérimenter, et puisque vous avez terminé les tutoriels DataCamp, reprenez maintenant `RStudio` et travaillez dans votre script.

1.3.4.2 Opérateurs de comparaison

Comme leur nom l'indique, ils permettent de comparer des valeurs ou des objets. Les principaux opérateurs de comparaison sont :

- `==` : égal à
- `!=` : différent de
- `>` : supérieur à
- `<` : inférieur à
- `>=` : supérieur ou égal à
- `<=` : inférieur ou égal à

Ainsi, on peut tester si 3 est égal à 5 :

```
3 == 5
```

```
[1] FALSE
```

La réponse est bien entendu `FALSE`. Est-ce que 3 est inférieur à 5 ?

```
3 < 5
```

```
[1] TRUE
```

La réponse est maintenant `TRUE`. Lorsque l'on utilise un opérateur de comparaison, la réponse est toujours soit vrai (`TRUE`), soit faux (`FALSE`).

Il est aussi possible de comparer des chaînes de charactères :

```
"Bonjour" == "Au revoir"
```

```
[1] FALSE
```

```
"Bonjour" >= "Au revoir"
```

```
[1] TRUE
```

Manifestement, “Bonjour” est supérieur ou égal à “Au revoir”. En fait, `R` utilise l’ordre alphabétique pour comparer les chaînes de caractères. Puisque dans l’alphabet, le “B” de “Bonjour” arrive après le “A” de “Au revoir”, pour `R`, “Bonjour” est supérieur à “Au revoir”.

Il est également possible d’utiliser ces opérateurs pour comparer un chiffre et un vecteur :

```
tailles_pop1 <- c(112, 28, 86, 14, 154, 73, 63, 48)
tailles_pop1 > 80
```

```
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

Ici, l’opérateur nous permet d’identifier quels éléments du vecteur `taille_pop1` sont supérieurs à 80. Il s’agit des éléments placés en première, troisième et cinquième positions.

Il est aussi possible de comparer 2 vecteurs qui contiennent le même nombre d’éléments :

```
tailles_pop2 <- c(114, 27, 38, 91, 54, 83, 33, 68)
tailles_pop1 > tailles_pop2
```

```
[1] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

Les comparaisons sont ici faites élément par élément. Ainsi, les observations 2, 3, 5 et 7 du vecteur `tailles_pop1` sont supérieures aux observations 2, 3, 5 et 7 du vecteur `tailles_pop2` respectivement.

Ces vecteurs de vrais/faux sont très utiles car ils peuvent permettre de compter le nombre d’éléments répondant à une certains condition :

```
sum(tailles_pop1 > tailles_pop2)
```

```
[1] 4
```

Lorsque l'on effectue une opération arithmétique (comme le calcul d'une somme ou d'une moyenne) sur un vecteur de vrais/faux, les `TRUE` sont remplacés par `1` et les `FALSE` par `0`. La somme nous indique donc le nombre de vrais dans un vecteur de vrais/faux, et la moyenne nous indique la proportion de vrais :

```
mean(tailles_pop1 > tailles_pop2)
```

```
[1] 0.5
```

Note : Attention, si les vecteurs comparés n'ont pas la même taille, un message d'avertissement est affiché :

```
tailles_pop3 <- c(43, 56, 92)
tailles_pop1
```

```
[1] 112  28  86  14 154  73  63  48
```

```
tailles_pop3
```

```
[1] 43 56 92
```

```
tailles_pop3 > tailles_pop1
```

```
Warning in tailles_pop3 > tailles_pop1: la taille d'un objet plus long n'est
pas multiple de la taille d'un objet plus court
```

```
[1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

Ici, `R` renvoie un résultat, accompagné d'un message d'avertissement qui nous indique que tout ne s'est probablement pas déroulé comme on le pensait. Dans un cas comme celui là, `R` va en effet *recycler* l'objet le plus court, ici `tailles_pop3` pour qu'une comparaison puisse être faite avec chaque élément de l'objet le plus long (ici, `tailles_pop1`). Ainsi, 43 est comparé à 112, 56 est comparé à 28 et 92 est comparé à 86. Puisque `tailles_pop3` ne contient plus d'éléments, ils sont recyclés, dans le même ordre : 43 est comparé à 14, 56 est comparé à 154, et ainsi de suite jusqu'à ce que tous les éléments de `tailles_pop1` aient été passés en revue.

Ce type de recyclage est très risqué car il est difficile de savoir ce qui a été comparé avec quoi. En travaillant avec des tableaux plutôt qu'avec des vecteurs, le problème est généralement évité puisque toutes les colonnes d'un `data.frame` contiennent le même nombre d'éléments.

⚠ Erreur ou avertissement ? 💣 ou ⚠ ?

Il ne faut pas confondre message d'erreur et message d'avertissement :

- Un message d'erreur commence généralement par `Error` ou `Erreur` et indique que `R` n'a pas compris ce que vous lui demandiez. Il n'a donc pas été en mesure de faire quoi que ce soit et votre commande n'a donc pas été exécutée. Vous devez absolument revenir à votre code et corriger la commande fautive car il y a fort à parier que si vous ne le faites pas, les commandes suivantes renverrons à leur tour un message d'erreur. Il est donc important de toujours revenir à la première erreur d'un script et de la corriger avant de passer à la suite.
- Un message d'avertissement commence généralement par `Warning` et vous indique que quelque chose d'inhabituel, ou de "non-optimal" a été réalisé. Un résultat a été produit, mais peut-être n'est-il pas conforme à ce que vous attendiez. La prudence est donc requise.

Dans les deux cas, un message explique de façon plus

ou moins claire ce qui a posé problème. Progresser dans la maîtrise du logiciel et du langage signifie en grande partie progresser dans la compréhension de la signification de ces messages parfois obscures. Pour progresser, il faut donc commencer par **lire attentivement ces messages**, et tenter de comprendre ce qu'ils veulent dire.

Dernière chose concernant les opérateurs de comparaison : la question des données manquantes. Dans `R` les données manquantes sont symbolisées par cette notation : `NA`, abréviation de "Not Available". Le symbole `NaN` (comme "Not a Number") est parfois aussi observé lorsque des opérations ont conduit à des indéterminations. Mais c'est plus rare et la plupart du temps, les `NaN`s peuvent être traités comme les `NA`s. L'un des problèmes des données manquantes est qu'il est nécessaire de prendre des précautions pour réaliser des comparaisons les impliquant :

```
3 == NA
```

```
[1] NA
```

On s'attend logiquement à ce que 3 ne soit pas considéré comme égal à `NA`, et donc, on s'attend à obtenir `FALSE`. Pourtant, le résultat est `NA`. La comparaison d'un élément quelconque à une donnée manquante fournit toujours une donnée manquante : la comparaison ne peut pas se faire, `R` n'a donc rien à retourner. C'est également le cas aussi lorsque l'on compare deux valeurs manquantes :

```
NA == NA
```

```
[1] NA
```

C'est en fait assez logique. Imaginons que j'ignore l'âge de Pierre et l'âge de Marie. Il n'y a aucune raison pour que leur âge soit le même, mais il est tout à fait possible qu'il le soit. C'est impossible à déterminer :

```
age_Pierre <- NA
age_Marie <- NA
age_Pierre == age_Marie
```

```
[1] NA
```

Mais alors comment faire pour savoir si une valeur est manquante puisqu'on ne peut pas utiliser les opérateurs de comparaison ? On utilise la fonction `is.na()` :

```
is.na(age_Pierre)
```

```
[1] TRUE
```

```
is.na(tailles_pop3)
```

```
[1] FALSE FALSE FALSE
```

D'une façon générale, le point d'exclamation permet de signifier à `R` que nous souhaitons obtenir le contraire d'une expression :

```
!is.na(age_Pierre)
```

```
[1] FALSE
```

```
!is.na(tailles_pop3)
```

```
[1] TRUE TRUE TRUE
```

Cette fonction nous sera très utile plus tard pour éliminer toutes les lignes d'un tableau contenant des valeurs manquantes.

1.3.4.3 L'utilisation des fonctions

Dans `R`, les fonctions sont des objets particuliers qui permettent d'effectuer des tâches très variées. Du calcul d'une moyenne à la création d'un graphique, en passant par la réalisation d'analyses statistiques complexes ou simplement l'affichage du chemin du répertoire de travail, tout, dans `R`, repose sur l'utilisation de fonctions. Vous en avez déjà vu un certain nombre :

Fonction	Pour quoi faire ?
`c()`	Créer des vecteurs
`class()`	Afficher ou modifier la classe d'un objet
`factor()`	Créer des facteurs
`getwd()`	Afficher le chemin du répertoire de travail
`head()`	Afficher les premiers éléments d'un objet
`is.na()`	Tester si un objet contient des valeurs manquantes
`mean()`	Calculer une moyenne
`names()`	Afficher ou modifier le nom des éléments d'un vecteur
`order()`	Ordonner les éléments d'un objet
`subset()`	Extraire une partie des éléments d'un objet
`sum()`	Calculer une somme
`tail()`	Afficher les derniers éléments d'un objet

Cette liste va très rapidement s'allonger au fil des séances. Je vous conseille donc vivement de tenir à jour une liste des fonctions décrites, avec une explication de leur fonctionnement et éventuellement un exemple de syntaxe.

Certaines fonctions ont besoin d'arguments (par exemple, la fonction `factor()`), d'autres peuvent s'en passer (par exemple, la fonction `getwd()`). Pour apprendre comment utiliser une fonction particulière, pour découvrir quels sont ses arguments possibles, quel est leur rôle et leur intérêt, la meilleure solution est de consulter l'aide de cette fonction. Il suffit pour cela de taper un `?` suivi du nom de la fonction :

```
?factor()
```

Toutes les fonctions et jeux de données disponibles dans `R` disposent d'un fichier d'aide similaire. Cela peut faire un peu peur au premier abord (tout est en anglais !), mais ces fichiers d'aide ont l'avantage d'être très complets, de fournir des exemples d'utilisation, et ils sont tous construits sur le même modèle. Vous avez donc tout intérêt à vous familiariser avec eux. Vous devriez d'ailleurs prendre l'habitude de consulter l'aide de chaque fonction qui vous pose un problème. Par exemple, le logarithme (en base 10) de 100 devrait faire 2, car 100 est égal à 10^2. Pourtant :

```
log(100)
```

```
[1] 4.60517
```

Que se passe-t'il ? Pour le savoir, il faut consulter l'aide de la fonction log :

```
?log()
```

Ce fichier d'aide nous apprend que par défaut, la syntaxe de la fonction `log()` est la suivante :

```
log(x, base = exp(1))
```

Par défaut, la base du logarithme est fixée à `exp(1)`. Nous avons donc calculé un logarithme népérien (en base e). Cette fonction prend donc 2 arguments :

1. `x` ne possède pas de valeur par défaut : il nous faut obligatoirement fournir quelque chose (la rubrique "Argument" du fichier d'aide nous indique que `x` doit être un vecteur numérique ou complexe) afin que la fonction puisse calculer un logarithme
2. `base` possède un argument par défaut. Si nous ne spécifions pas nous même la valeur de `base`, elle sera fixée à sa valeur par défaut, c'est à dire `exp(1)`.

Pour calculer le logarithme de 100 en base 10, il faut donc taper, au choix, l'une de ces 3 expressions :

```
log(x = 100, base = 10)
```

```
[1] 2
```

```
log(100, base = 10)
```

```
[1] 2
```

```
log(100, 10)
```

```
[1] 2
```

Le nom des arguments d'une fonction peut être omis tant que ses arguments sont indiqués dans l'ordre attendu par la fonction (cet ordre est celui qui est précisé à la rubrique "Usage" du fichier d'aide de la fonction). Il est possible de modifier l'ordre des arguments d'une fonction, mais il faut alors être parfaitement explicite et utiliser les noms des arguments tels que définis dans le fichier d'aide.

Ainsi, pour calculer le logarithme de 100 en base 10, on ne peut pas taper :

```
log(10, 100)
```

```
[1] 0.5
```

car cela revient à calculer le logarithme de 10 en base 100. On peut en revanche taper :

```
log(base = 10, x = 100)
```

```
[1] 2
```

1.4 Les packages additionels

Une source de confusion importante pour les nouveaux utilisateurs de `R` est la notion de package. Les packages étendent les fonctionnalités de `R` en fournissant des fonctions, des données et de la documentation supplémentaires et peuvent être téléchargés gratuitement sur Internet. Ils sont écrits par une communauté mondiale d'utilisateurs de `R`. Par exemple, parmi les plus de 18000 packages disponibles à l'heure actuelle, nous utiliseront fréquemment :

- Le package `ggplot2` pour la visualisation des données dans le Chapitre 3
- Le package `dplyr` pour manipuler des tableaux de données dans le Chapitre 4

Une bonne analogie pour les packages `R` : ils sont comme les apps que vous téléchargez sur un téléphone portable. `R` est comme un nouveau téléphone mobile. Il est capable de faire certaines choses lorsque vous l'utilisez pour la première fois, mais il ne sait pas tout faire. Les packages sont comme les apps que vous pouvez télécharger dans l'App Store et Google Play. Pour utiliser un package, comme pour utiliser Instagram, vous devez :

1. Le télécharger et l'installer. Vous ne le faites qu'une fois grâce à la commande `install.packages()`
2. Le charger (en d'autres termes, l'ouvrir) en utilisant la commande `library()` à chaque nouvelle session de travail

Donc, tout comme vous ne pouvez commencer à partager des photos avec vos amis sur Instagram que si vous installez d'abord l'application et que vous l'ouvrez, vous ne pouvez accéder aux données et fonctions d'un package `R` que si vous installez d'abord le package et le chargez avec la fonction `library()`. Passons en revue ces 2 étapes.

1.4.1 Installation d'un package

Il y a deux façons d'installer un package. Par example, pour installer le package `ggplot2` :

1. **Le plus simple** : Dans le quart inférieur droit de l'interface de `Rstudio` :
    a) Cliquez sur l'onglet "Packages"
    b) Cliquez sur "Install"
    c) Tapez le nom du package dans le champ "Packages (separate multiple with space or comma):" Pour notre exemple, tapez `ggplot2`
    d) Cliquez sur "Install"
2. **Métode alternative** : Dans la console, tapez `install.packages("ggplot2")` (vous devez inclure les guillemets).

En procédant de l'une ou l'autre façon, installez également les packages suivants : `tidyverse` et `palmerpenguins`. Le `tidyverse` est un "méta-package", qui permet en fait d'installer de nombreux packages en une seule commande, dont `ggplot2`, `tidyr`, `dplyr`, `magrittr` et bien d'autres. Le package `palmerpenguins` contient un jeu de données dont nous nous servirons copieusement dans les chapitres suivants.

> **i** Note : `install.packages()`
>
> Un package doit être installé une fois seulement sur un ordinateur, sauf si une version plus récente est disponible et que vous souhaitez mettre à jour ce package. Il n'est donc pas nécessaire de laisser ces commandes dans votre script. Sinon, vous risquez de ré-installer les packages à chaque nouvelle session de travail, ce qui est inutile et consomme inutilement de la bande passante, des ressources numériques, et donc, du carbone… 🌡🌎

1.4.2 Charger un package en mémoire

Après avoir installé un package, vous pouvez le charger en utilisant la fonction `library()`. Par exemple, pour charger `ggplot2` et `dplyr` tapez ceci dans la console :

```
library(ggplot2)
library(dplyr)
```

Puisque ces packages font partie du `tidyverse`, on aurait pu les charger tous les deux (et d'autres) en une seule étape en tapant :

```
library(tidyverse)
```

Quand vous exécutez une commande, si vous voyez un message d'erreur commençant par :

```
Error: could not find function...
```

c'est probablement parce que vous tentez d'utiliser une fonction qui fait partie d'un package que vous n'avez pas chargé. Pour corriger l'erreur, il suffit donc de charger le package approprié avec la commande `library()`.

> **Note : `library()`**
>
> Vous devrez charger à nouveau chaque package que vous souhaitez utiliser **à chaque fois que vous ouvrirez une nouvelle session de travail dans `RStudio`** (à chaque nouveau démarrage du logiciel, donc). C'est une erreur fréquente pour les débutants. Pour l'éviter, pensez bien à intégrer, tout en haut de votre script, les commandes `library()` nécessaires pour chaque package que vous comptez utiliser.

1.5 Exercice

Dans votre dossier de travail, créez un nouveau script que vous nommerez `ExoDiamonds.R`. Vous prendrez soin d'ajouter autant de commentaires que nécessaire dans votre script afin de le structurer correctement.

1. Téléchargez (si besoin) et chargez le package `ggplot2`
2. Chargez le jeu de données `diamonds` grâce à la commande `data(diamonds)`
3. Déterminez le nombre de lignes et de colonnes de ce tableau nommé `diamonds`

4. Créez un nouveau tableau que vous nommerez `diamants_chers` qui contiendra uniquement les informations des diamants dont le prix est supérieur ou égal à $15000.
5. Combien de diamants coûtent $15000 ou plus ?
6. Cela représente quelle proportion du jeu de données de départ ?
7. Triez ce tableau par ordre de prix décroissants et affichez les informations des 20 diamants les plus chers.

2 Explorez votre premier jeu de données

2.1 Préambule

Mettons en pratique tout ce que nous avons appris pour commencer à explorer un jeu de données réel. Les données nous parviennent sous différents formats, des images au texte en passant par des tableaux de chiffres. Tout au long de ce document, nous nous concentrerons sur les ensembles de données qui peuvent être stockés dans une feuille de calcul, car il s'agit de la manière la plus courante de collecter des données dans de nombreux domaines. N'oubliez pas ce que nous avons appris dans la Section 1.3.4.1 : ces ensembles de données de type "tableurs" sont appelés `data.frame` dans `R`, et nous nous concentrerons sur l'utilisation de ces objets tout au long de ce livre. S'il est évidemment possible d'importer dans `R` des données stockées dans des fichiers Excel ou des fichiers textes, nous allons dans un premier temps faire plus simple : nous travaillerons avec des données déjà disponibles dans un packages que nous avons installé dans la Section 1.4.

Ainsi, commençons par charger les packages nécessaires pour ce chapitre (cela suppose que vous les ayez déjà installés ; relisez la Section 1.4 pour plus d'informations sur l'installation et le chargement des packages `R` si vous ne l'avez pas déjà fait). Au début de chaque chapitre, nous aurons systématiquement besoin de charger quelques packages. Donc n'oubliez pas de les installer au préalable si besoin.

```
# Pensez à installer ces packages avant de les charger si besoin
library(dplyr)
library(palmerpenguins)
```

2.2 Le package `palmerpenguins`

Ce package (Horst, Hill, et Gorman 2022) contient un jeu de données collectées par Kristen Gorman (membre du "Long Term Ecological Research Network' ') et la station de Palmer en Antarctique (Gorman, Williams, et Fraser 2014). Les données contiennent des informations au sujet de 330 individus appartenant à 3 espèces de manchots (voir Figure 2.1) étudiés sur 3 îles de l'archipel de Palmer, an Antarctique. Ces espèces ont fait l'objet de nombreuses études comparatives, notamment afin de déterminer comment elles utilisent le milieu pour acquérir des ressources. Puisque ces 3 espèces sont proches sur le plan phylogénétique et qu'elles occupent le même habitat, la question de la compétition inter-spécifique, pour l'espace et les ressources, se pose tout naturellement.

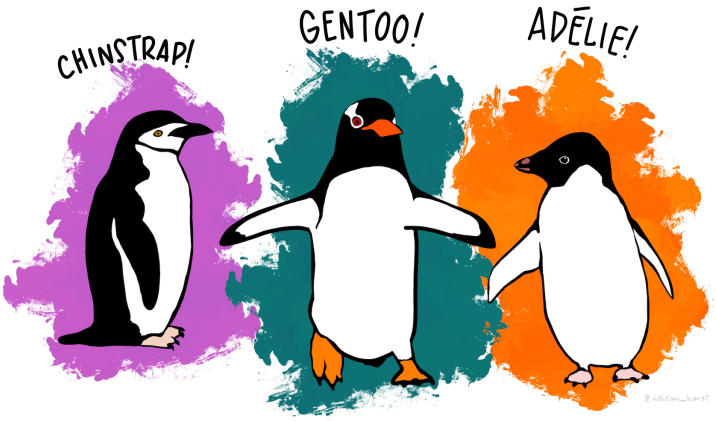


Figure 2.1: Les 3 espèces de manchots de l'archipel de Palmer. Illustration : Allison Horst

2.3 Le data frame `penguins`

Nous allons commencer par explorer le jeu de données `penguins` qui est inclus avec le package `palmerpenguins` afin de nous faire une idée de sa structure. Dans votre script, tapez la commande suivante et exécutez la dans la console (selon les réglages de `RStudio` et *la largeur de votre console*, l'affichage peut varier légèrement) :

```
penguins
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Essayons de décrypter cet affichage :

- `A tibble: 344 x 8` : un tibble est un `data.frame` amélioré. Il a toutes les caractéristiques d'un `data.frame`, (tapez `class(penguins)` pour vous en convaincre), mais en plus, il a quelques propriétés intéressantes sur lesquelles nous reviendrons plus tard. Ce `tibble` possède donc :
  - 344 lignes
  - 8 colonnes, qui correspondent aux variables. Dans un `tibble`, les observations sont toujours en lignes et les variables en colonnes
- `species`, `island`, `bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`… sont les noms des colonnes, c'est à dire les variables de ce jeu de données
- Nous avons ensuite les 10 premières lignes du tableau
- `... with 334 more rows, and abbreviated variable names...`, nous indique que 334 lignes ne logent pas à l'écran et que le nom de certains variables a été abrégé afin de permettre un affichage plus clair. Ces données font toutefois partie intégrante du tableau `penguins`

- les noms complets de toutes les variables abrégées sont également indiqués

Cette façon d'afficher les tableaux est spécifique des `tibble`s. Vous noterez que le type de chaque variable est indiqué entre `<...>`, juste sous les noms de colonnes. Voici certains des types de données que vous pourrez rencontrer :

- `<int>` : nombres entiers ("integers")
- `<dbl>` : nombres réels ("doubles")
- `<chr>` : caractères ("characters")
- `<fct>` : facteurs ("factors")
- `<ord>` : facteurs ordonnés ("ordinals")
- `<lgl>` : logiques (colonne de vrais/faux : "logical")
- `<date>` : dates
- `<time>` : heures
- `<dttm>` : combinaison de date et d'heure ("date time")

Cette façon d'afficher le contenu d'un tableau permet d'y voir (beaucoup) plus clair que l'affichage classique d'un `data.frame`. Malheureusement, ce n'est pas toujours suffisant. Voyons quelles sont les autres méthodes permettant d'explorer un `data.frame`.

2.4 Explorer un `data.frame`

Parmi les nombreuses façons d'avoir une idée des données contenues dans un `data.frame` tel que `penguins`, on présente ici 3 fonctions qui prennent le nom du `data.frame` en guise d'argument, et un opérateur :

- la fonction `View()` intégrée à `RStudio`. C'est celle que vous utiliserez le plus souvent. Attention, elle s'écrit avec un "V" majuscule
- la fonction `glimpse()` chargée avec le package `dplyr`. Elle est très similaire à la fonction `str()` découverte dans les tutoriels de DataCamp
- l'opérateur `$` permet d'accéder à une unique variable d'un `data.frame`
- la fonction `skim()` du package `skimr` permet d'obtenir un résumé complet mais très synthétique et visuel des variables d'un `data.frame`

2.4.1 `View()`

Tapez `View(penguins)` dans votre script et exécutez la commande. Un nouvel onglet contenant ce qui ressemble à un tableur doit s'ouvrir.

> Quizz : à quoi correspondent chacune des lignes de ce tableau ?
>
> a. aux données d'une espèce
> b. aux données d'une île
> c. aux données d'un individu
> d. aux données d'une population (plusieurs manchots à la fois)

Ici, vous pouvez donc explorer la totalité du tableau, passer chaque variable en revue, et même appliquer des filtres pour ne visualiser qu'une partie des données. Par exemple, essayez de déterminer combien d'individus sont issus de l'île "Biscoe".

Ce tableau n'est pas facile à manipuler. Il est impossible de corriger des valeurs, et lorsque l'on applique des filtres, il est impossible de récupérer uniquement les données filtrées. Nous verrons plus tard comment les obtenir en tapant des commandes simples dans un script. La seule utilité de ce tableau est donc l'exploration visuelle des données.

2.4.2 `glimpse()`

La seconde façon d'explorer les données contenues dans un tableau est d'utiliser la fonction `glimpse()` après avoir chargé le package `dplyr` :

```
glimpse(penguins)
```

```
Rows: 344
Columns: 8
$ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
$ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
$ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
$ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
$ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
```

```
$ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
$ sex               <fct> male, female, female, NA, female, male, female, male~
$ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

Ici, les premières observations sont présentées en lignes pour chaque variable du jeu de données. Là encore, le type de chaque variable est précisé. Essayez d'identifier 3 variables catégorielles. À quoi correspondent-elles ? En quoi sont-elles différentes des variables numériques ?

2.4.3 L'opérateur $

L'opérateur `$` permet d'accéder à une unique variable grâce à son nom. Par exemple on peut accéder à toutes les données concernant les noms d'espèces (variable `species` du tableau `penguins`) en tapant :

```
penguins$species
```

```
  [1] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
  [8] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [15] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [22] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [29] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [36] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [43] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [50] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [57] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [64] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [71] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [78] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [85] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [92] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [99] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[106] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[113] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[120] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[127] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[134] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[141] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[148] Adelie    Adelie    Adelie    Adelie    Adelie    Gentoo    Gentoo
[155] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
```

```
[162] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[169] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[176] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[183] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[190] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[197] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[204] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[211] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[218] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[225] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[232] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[239] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[246] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[253] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[260] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[267] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[274] Gentoo    Gentoo    Gentoo    Chinstrap Chinstrap Chinstrap Chinstrap
[281] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[288] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[295] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[302] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[309] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[316] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[323] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[330] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[337] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[344] Chinstrap
Levels: Adelie Chinstrap Gentoo
```

Cela nous permet de récupérer les données sous la forme d'un vecteur ou, comme ici, d'un facteur. Attention toutefois, le tableau `penguins` contient beaucoup de lignes. Récupérer une variable grâce à cet opérateur peut rapidement saturer la console. Nous serons amenés à manipuler des tableaux contenant plusieurs dizaines ou centaines de milliers de lignes. C'est le cas du tableau `diamonds` du package `ggplot2` que vous avez découvert dans les exercice de la Section 1.5.

Si, par exemple, vous souhaitez extraire les données relatives à la clarté des diamants (colonne `clarity`) du tableau `diamonds`, vous pouvez taper ceci :

```
library(ggplot2)
diamonds$clarity
```

Le résultat est pour le moins indigeste ! Lorsqu'un tableau contient de nombreuses lignes, c'est rarement une bonne idée de transformer l'une de ses colonnes en vecteur. Dans la mesure du possible, les données d'un tableau doivent rester dans le tableau.

2.4.4 `skim()`

Pour utiliser la fonction `skim()`, vous devez au préalable installer le package `skimr` :

```
install.packages("skimr")
```

Ce package est un peu "expérimental" et il se peut que l'installation pose problème. Si un message d'erreur apparaît lors de l'installation, procédez comme suit :

1. Quittez `RStudio` (sans oublier de sauvegarder votre travail au préalable)
2. Relancez `RStudio` et dans la console, tapez ceci :

```
install.packages("rlang")
```

3. Tentez d'installer `skimr` à nouveau.
4. Exécutez à nouveau tout votre script afin de retrouver votre travail dans l'état où il était avant de quitter `RStudio`.

Si l'installation de `skimr` s'est bien passée, vous pouvez maintenant taper ceci :

```
library(skimr)
skim(penguins)
```

Table 2.1: Data summary

Name	penguins
Number of rows	344
Number of columns	8

Column type frequency:	
factor	3
numeric	5
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
species	0	1.00	FALSE	3	Ade: 152, Gen: 124, Chi: 68
island	0	1.00	FALSE	3	Bis: 168, Dre: 124, Tor: 52
sex	11	0.97	FALSE	2	mal: 168, fem: 165

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
bill_length_mm	2	0.99	43.92	5.46	32.1	39.23	44.45	48.5	59.6	
bill_depth_mm	2	0.99	17.15	1.97	13.1	15.60	17.30	18.7	21.5	
flipper_length_mm	2	0.99	200.92	14.06	172.0	190.00	197.00	213.0	231.0	
body_mass_g	2	0.99	4201.75	801.95	2700.0	3550.00	4050.00	4750.0	6300.0	
year	0	1.00	2008.03	0.82	2007.0	2007.00	2008.00	2009.0	2009.0	

Nous aurons l'occasion de revenir en détail sur la signification de tous ces indices au semestre prochain. À ce stade, retenez que cette fonction `skim()` permet d'accéder à un résumé très détaillé de chaque variable d'un jeu de données. Par exemple, on apprend ici que la masse corporelle moyenne des manchots de l'ensemble du jeu de données vaut 4201.75 grammes (ligne `body_mass_g`, colonne `mean`), avec un écart-type de 0.82 grammes (colonne `sd`), et que la masse de 2 individus est manquante (colonne `n_missing`). Cette fonction nous sera donc très utile au semestre prochain lorsque nous aborderons la question des statistiques descriptives.

2.4.5 Les fichiers d'aide

Une fonctionnalité particulièrement utile de `R` est son système d'aide. On peut obtenir de l'aide au sujet de n'importe quelle fonction et de n'importe quel jeu de données en tapant un "`?`" immédiatement suivi du nom de la fonction ou de l'objet.

Par exemple, examinez l'aide du jeu de données `penguins` :

```
?penguins
```

Vous devriez absolument prendre l'habitude d'examiner les fichiers d'aide des fonctions ou jeux de données pour lesquels vous avez des questions. Ces fichiers sont très complets, et même s'il peuvent paraître impressionnants au premier abord, ils sont tous structurés sur le même modèle et vous aideront à comprendre comment utiliser les fonctions, quels sont les arguments possibles, à quoi ils servent et comment les utiliser.

Prenez le temps d'examiner le fichier d'aide du jeu de données `penguins`. Avant de passer à la suite, assurez-vous d'avoir compris à quoi correspondent chacune des 8 variables de ce tableau.

2.5 Exercice

Consultez l'aide du jeu de données `diamonds` du package `ggplot2`.

- Quel est le code de la couleur la plus prisée ?
- Quel est le code de la moins bonne clarté ?
- À quoi correspond la variable `z` ?
- En quoi la variable `depth` est-elle différente de la variable `z` ?

3 Visualiser des données avec `ggplot2`

3.1 Préambule

Dans le Chapitre 1 et le Chapitre 2, vous avez découvert les concepts essentiels qu'il est important de maîtriser avant de commencer à explorer en détail des données dans `R`. Les éléments de syntaxe abordés dans la Section 1.3 sont nombreux et vous n'avez probablement pas tout retenu. C'est pourquoi je vous conseille de garder les tutoriels de DataCamp à portée de main afin de pouvoir refaire les parties que vous maîtrisez le moins. Ce n'est qu'en répétant plusieurs fois ces tutoriels que les choses seront vraiment comprises et que vous les retiendrez. Ainsi, si des éléments de code présentés ci-dessous vous semblent obscurs, revenez en arrière : toutes les réponses à vos questions se trouvent probablement dans les chapitres précédents.

Après la découverte des bases du langage `R`, nous abordons maintenant les parties de ce livre qui concernent la "science des données" (ou "Data Science" pour nos amis anglo-saxons). Nous allons voir dans ce chapitre qu'outre les fonctions `View()` et `glimpse()`, l'exploration visuelle *via* la représentation graphique des données est un moyen indispensable et très puissant pour comprendre ce qui se passe dans un jeu de données.

> **!** Important
>
> La visualisation de vos données devrait toujours être un **préalable indispensable** à toute analyse statistique.

La visualisation des données est en outre un excellent point de départ quand on découvre la programmation sous `R`, car ses bénéfices sont clairs et immédiats : vous pouvez créer des graphiques élégants et informatifs qui vous aident à comprendre

les données. Dans ce chapitre, vous allez donc plonger dans l'art de la visualisation des données, en apprenant la structure de base des graphiques réalisés avec `ggplot2` qui permettent de transformer des données numériques et catégorielles en graphiques.

Toutefois, la visualisation seule ne suffit généralement pas. Il est en effet souvent nécessaire de transformer les données pour produire des représentations plus parlantes. Ainsi, dans le Chapitre 4, vous découvrirez les fonctions clés qui vous permettront de sélectionner des variables importantes, de filtrer des observations, de créer de nouvelles variables, ou d'en modifier la forme.

Ce n'est qu'en combinant les transformations de données et représentations graphiques d'une part, avec votre curiosité et votre esprit critique d'autre part, que vous serez véritablement en mesure de réaliser une analyse exploratoire de vos données à la fois utile et pertinente. C'est la seule façon d'identifier des questions intéressantes sur vos données, afin de tenter d'y répondre par les analyses statistiques et la modélisation qui seront abordées lors des prochains semestres.

3.2 Prérequis

Dans ce chapitre, nous aurons besoin des packages suivants :

```
library(tidyverse)
library(palmerpenguins)
library(nycflights13)
library(gapminder)
library(scales)
```

Si ce n'est pas déjà fait, pensez à les installer avant de les charger en mémoire.

Au niveau le plus élémentaire, les graphiques permettent de comprendre comment les variables se comparent en termes de tendance centrale (à quel endroit les valeurs ont tendance à être localisées, regroupées) et leur dispersion (comment les données varient autour du centre). La chose la plus importante à savoir sur les graphiques est qu'ils doivent être créés

pour que votre public (le professeur qui vous évalue, le collègue avec qui vous collaborez, votre futur employeur, etc.) comprenne bien les résultats et les informations que vous souhaitez transmettre. Il s'agit d'un exercice d'équilibriste : d'une part, vous voulez mettre en évidence autant de relations significatives et de résultats intéressants que possible, mais de l'autre, vous ne voulez pas trop en inclure, afin d'éviter de rendre votre graphique illisible ou de submerger votre public. Tout comme n'importe quel paragraphe de document écrit, un graphique doit permettre de **communiquer un message** (une idée forte, un résultat marquant, une hypothèse nouvelle, etc).

Comme nous le verrons, les graphiques nous aident également à repérer les tendances extrêmes et les valeurs aberrantes dans nos données. Nous verrons aussi qu'une façon de faire, assez classique, consiste à comparer la distribution d'une variable quantitative pour les différents niveaux d'une variable catégorielle.

Objectifs

Dans ce chapitre, vous apprendrez à :

1. faire différents types de graphiques exploratoires avec le package `ggplot2`
2. choisir le ou les graphiques appropriés selon la nature des variables dont vous disposez ou que vous souhaitez mettre en relation
3. mettre vos graphiques en forme pour les intégrer dans vos rapports ou compte-rendus de TP

3.3 La grammaire des graphiques

Les lettres `gg` du package `ggplot2` sont l'abréviation de "**g**rammar of **g**raphics" : la grammaire des graphiques. De la même manière que nous construisons des phrases en respectant des règles grammaticales précises (usage des noms, des verbes, des sujets et adjectifs…), la grammaire des graphiques établit un certain nombre de règles permettant de construire des graphiques : elle précise les composants d'un

graphique en suivant le cadre théorique défini par Wilkinson (2005).

3.3.1 Éléments de la grammaire

En bref, la grammaire des graphiques nous dit que :

> Un graphique est l'association (`mapping`) de données/variables (`data`) à des attributs esthétiques (`aes`thetics) d'objets géométriques (`geom`etric objects).

Pour clarifier, on peut disséquer un graphique en 3 éléments essentiels :

1. `data` : le jeu de données contenant les variables que l'on va associer à des objets géométriques. Pour `ggplot2` les données doivent obligatoirement être stockées dans un `data.frame` ou un `tibble`
2. `geom` : les objets géométriques en question. Cela fait référence aux types d'objets que l'on peut observer sur le graphique (des points, des lignes, des barres, etc.)
3. `aes` : les attributs esthétiques des objets géométriques présents sur le graphique. Par exemple, la position sur les axes `x` et `y`, la couleur, la taille, la transparence, la forme, etc. Chacun de ces attributs esthétiques peut-être associé à une variable de notre jeu de données.

Examinons un exemple pour bien comprendre.

3.3.2 Gapminder

En février 2006, un statisticien du nom de Hans Rosling a donné un TED Talk intitulé "The best stats you've ever seen". Au cours de cette conférence, Hans Rosling présente des données sur l'économie mondiale, la santé et le développement des pays du monde. Les données sont disponibles sur ce site et dans le package `gapminder`.

Pour l'année 2007, le jeu de données contient des informations pour 142 pays. Examinons les premières lignes de ce jeu de données :

Table 3.1: Les 6 premières lignes du jeu de données `gapminder` pour l'année 2007.

Country	Continent	Life Expectancy	Population	GDP per Capita
Afghanistan	Asia	43.828	31889923	974.5803
Albania	Europe	76.423	3600523	5937.0295
Algeria	Africa	72.301	33333216	6223.3675
Angola	Africa	42.731	12420476	4797.2313
Argentina	Americas	75.320	40301927	12779.3796
Australia	Oceania	81.235	20434176	34435.3674

Pour chaque ligne, les variables suivantes sont décrites :

- `Country` : le pays
- `Continent` : le continent
- `Life Expectancy` : espérance de vie à la naissance
- `Population` : nombre de personnes vivant dans le pays
- `GDP per Capita` : produit intérieur brut (PIB) par habitant en dollars américains. GDP est l'abréviation de "Growth Domestic Product". C'est un indicateur de l'activité économique d'un pays, parfois utilisé comme une approximation du revenu moyen par habitant.

Examinons maintenant la Figure 3.1 qui représente ces variables pour chacun des 142 pays de ce jeu de données (notez l'utilisation de la notation scientifique dans la légende, et de l'échelle logarithmique de l'axe des abscisses).

Si on décrypte ce graphique du point de vue de la grammaire des graphiques, on voit que :

- la variable `GDP per Capita` est associée à l'`aes`thetic `x` de la position des points
- la variable `Life Expectancy` est associée à l'`aes`thetic `y` de la position des points
- la variable `Population` est associée à l'`aes`thetic `size` (taille) des points
- la variable `Continent` est associée à l'`aes`thetic `color` (couleur) des points

Ici, l'objet géométrique (ou `geom`) qui représente les données est le point. Les données (ou `data`) sont contenues dans le tableau `gapminder` et chacune de ces variables est associée (`mapping`) aux caractéristiques esthétiques des points.

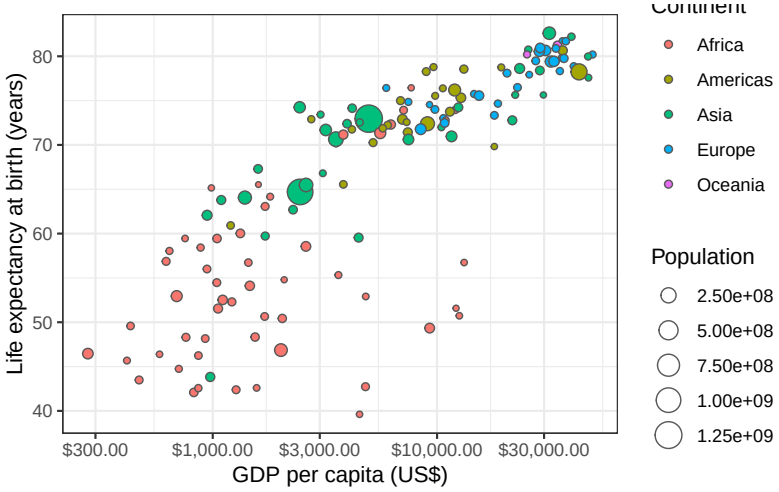


Figure 3.1: Espérance de vie en fonction du PIB par habitant en 2007.

3.3.3 Autres éléments de la grammaire des graphiques

Outre les éléments indispensables évoqués ici (`data`, `mapping`, `aes`, et `geom`), il existe d'autres aspects de la grammaire des graphiques qui permettent de contrôler l'aspect des graphiques. Ils ne sont pas toujours indispensables. Nous en verrons néanmoins quelque-uns particulièrement utiles :

- `facet` : c'est un moyen très pratique de scinder le jeu de données en plusieurs sous-groupes et de produire automatiquement un graphique pour chacun d'entre eux.
- `position` : permet notamment de modifier la position des barres d'un barplot.
- `labs` : permet de définir les titres, sous-titres et légendes des axes d'un graphique
- `theme` : permet de modifier l'apect général des graphiques en appliquant des thèmes prédéfinis ou en modifiant certains aspects de thèmes existants

3.3.4 Le package `ggplot2`

Comme indiqué plus haut, le package `ggplot2` (Wickham et al. 2024) permet de réaliser des graphiques dans `R` en respectant les principes de la grammaire des graphiques. Vous

avez probablement remarqué que depuis le début de la section Section 3.3, beaucoup de termes sont écrits dans la police réservée au `code` informatique. C'est parce que les éléments de la grammaire des graphiques sont tous précisés dans la fonction `ggplot()` qui demande, au grand minimum, que les éléments suivants soient spécifiés :

- le nom du `data.frame` contenant les variables qui seront utilisées pour le graphique. Ce nom correspond à l'argument `data` de la fonction `ggplot()`.
- l'association des variables à des attributs esthétiques. Cela se fait grâce à l'argument `mapping` et la fonction `aes()`

Après avoir spécifié ces éléments, on ajoute des couches supplémentaires au graphique grâce au signe `+`. La couche la plus essentielle à ajouter à un graphique, est une couche contenant un élément géométrique, ou `geom` (par exemple des points, des lignes ou des barres). D'autres couches peuvent s'ajouter pour spécifier des titres, des `facet`s ou des modifications des axes et des thèmes du graphique.

Dans le cadre de ce cours, nous nous limiterons aux 5 types de graphiques suivants :

1. les nuages de points
2. les graphiques en lignes
3. les histogrammes
4. les diagrammes bâtons
5. les boîtes à moustaches (mais nous en dirons plus à ce sujet au semestre prochain)

3.3.5 Votre premier graphique

Reprenons maintenant le jeu de données `penguins` :

```
penguins
```

```
# A tibble: 344 x 8
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
```

```
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Comme évoqué plus haut, il s'agit d'un `tibble`. Plusieurs de ses variables concernent la biométrie des manchots, en particulier de son bec (voir Figure 3.2).

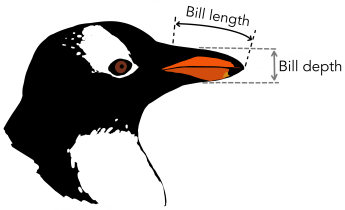


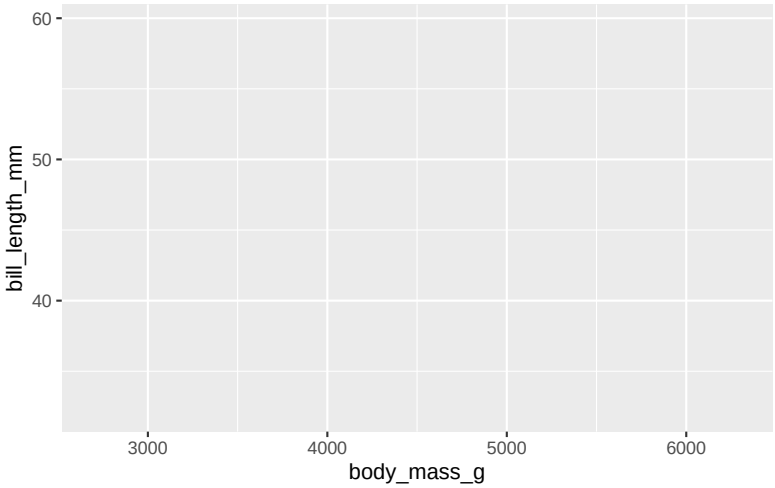
Figure 3.2: Morphométrie du bec des manchots. Illustration de Allison Horst

Supposons qu'on cherche à déterminer si la longueur du bec des manchots est proportionnelle à leur masse. Pour produire un graphique permettant de le déterminer, nous avons besoin des éléments suivants :

1. `data` : le tableau `penguins`
2. un objet géométrique, ici, des points (`geom_point()`) puisque nous disposons de 2 variables numériques (plus de détails à ce sujet plus bas)
3. l'association de certaines variables du jeu de données (ici, `body_mass_g` et `bill_length_mm`) à certaines caractéristiques esthétiques du graphiques (ici, la position sur les axes des `x` et des `y`), grâce à l'argument `mapping` et la fonction `aes()`.

Concrètement, voilà le code qu'il faut taper dans votre script :

```r
ggplot(data = penguins, mapping = aes(x = body_mass_g, y = bill_length_mm))
```



Cette première ligne de code permet de faire plusieurs choses :

1. on indique à `R` qu'on souhaite faire un graphique (avec la fonction `ggplot()`)
2. on indique à `R` que les données sont contenues dans l'objet `penguins` avec `data = penguins`
3. on associe (avec `mapping =` la variable `body_mass_g` à l'axe des `x` et la variable `bill_length_mm` à l'axe des `y`. On fait cela grâce à `aes(x = body_mass_g, y = bill_length_mm)`

Cette commande génère la première couche du graphique. Il n'y a pas encore de données car nous n'avons pas indiqué quel type d'objet géométrique nous souhaitons afficher, mais la fenêtre graphique est bel et bien créée, les axes apparaissent, ils sont légendés et leur échelle est adaptée aux variables du tableau `penguins` que nous avons sélectionnées. Pour terminer le graphique, il nous faut donc ajouter une seconde couche, celle de l'objet géométrique :

```r
ggplot(data = penguins, mapping = aes(x = body_mass_g, y = bill_length_mm)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

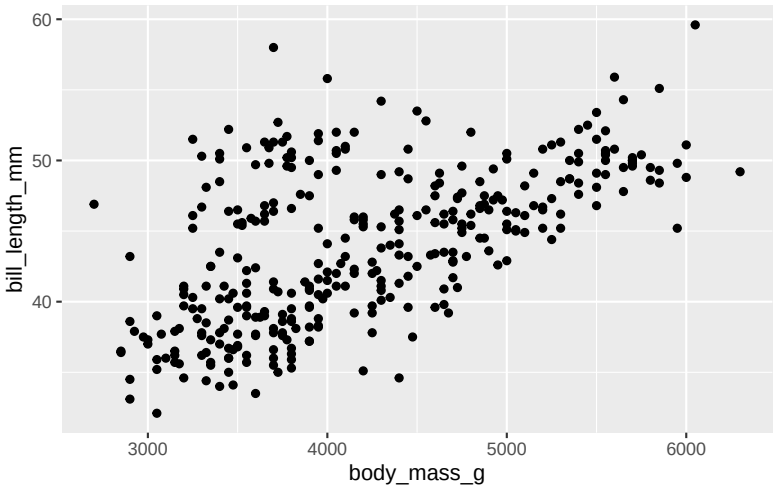


Figure 3.3: Relation entre masse corporelle et longueur du bec chez les manchots de l'archipel de Palmer

Au moment de produire ce graphique, `R` nous indique que 2 lignes du tableau `penguins` ne figurent pas sur ce graphique car elles possèdent des données manquantes (`NA`), pour l'une et/ou l'autre des variables que nous avons sélectionnées. La fonction `geom-point()` est donc incapable de les placer sur le graphique.

Vous avez donc ici un premier exemple de graphique très simple. Il est loin d'être parfait (à minima, le titre des axes devrait être modifié), mais il a le mérite de vous présenter la syntaxe que vous devrez utiliser pour produire presque tous les graphiques qui vous seront utiles avec `ggplot2`. En outre, on peut percevoir qu'il semble exister une relation positive (mais imparfaite) entre longueur des becs et masse des individus. Il faut toutefois être prudent car nous avons ici utilisé toutes les données disponibles (donc les données des 3 espèces à la fois), ce qui est loin d'être pertinent.

En résumé

- Au sein de la fonction `ggplot()`, on spécifie 2 composants de la grammaire des graphiques :
    1. le nom du tableau contenant les données grâce à l'argument `data = penguins`
    2. l'association (`mapping`) des variables du ta-

bleau de données à des caractéristiques esthétiques (`aes()`) en précisant `aes(x = body_mass_g, y = bill_length_mm)` :
  - la variable `body_mass_g` est associée à l'esthétique de position `x`
  - la variable `bill_length_mm` est associée à l'esthétique de position `y`

- On ajoute une couche au graphique `ggplot()` grâce au symbole `+`. La couche en question précise le troisième élément indispensable de la grammaire des graphiques : l'objet `geom`étrique. Ici, les objets sont des `point`s. On le spécifie grâce à la fonction `geom_point()`.

Quelques remarques concernant les couches :

- Notez que le signe `+` est placé *à la fin de la ligne*. Vous recevrez un message d'erreur si vous le placez au début.
- Quand vous ajoutez une couche à un graphique, je vous encourage vivement à presser la touche `enter` de votre clavier juste après le symbole `+`. Ainsi, le code correspondant à chaque couche sera sur une ligne distincte, ce qui augmente considérablement la lisibilité de votre code.
- Comme indiqué dans la Section 1.3.4.3, tant que les arguments d'une fonction sont spécifiés dans l'ordre, on peut se passer d'écrire leur nom. Ainsi, les deux blocs de commande suivants produisent exactement le même résultat :

```
# Le nom des arguments est précisé
ggplot(data = penguins, mapping = aes(x = body_mass_g, y = bill_length_mm)) +
  geom_point()

# Le nom des arguments est omis
ggplot(penguins, aes(x = body_mass_g, y = bill_length_mm)) +
  geom_point()
```

3.3.6 Exercices

1. Donnez une raison pratique expliquant pourquoi les variables `body_mass_g` et `bill_length_mm` ont une relation positive
2. Quelles variables (pas nécessairement dans le tableau `penguins`) pourraient avoir une corrélation négative (relation négative) avec `body_mass_g` ? Pourquoi ? Rappelez-vous que nous étudions ici des variables numériques.
3. Citez les éléments de ce graphique/de ces données qui vous sautent le plus aux yeux ?
4. Créez un nouveau nuage de points en utilisant d'autres variables du jeu de données `penguins`

3.4 Quel graphique dans quelle situation ?

Il n'est pas possible de faire n'importe quel type de graphique dans n'importe quelle situation. Selon le nombre de variables dont on dispose ou que l'on souhaite examiner, et selon la nature de ces variables (numériques et/ou catégorielles), le choix des types de graphiques possibles sera limité. Par exemple, les diagrammes bâtons sont réservés aux variables catégorielles, alors que les histogrammes sont possibles uniquement avec les variables numériques continues. Néanmoins, dans certaines situations, plusieurs choix de graphiques seront possibles, et vous aurez donc une certaine liberté. Vos choix seront alors guidés par les objectifs que vous souhaiterez atteindre grâce aux graphiques, ainsi que par vos préférences.

Objectifs

Dans la suite de ce chapitre, nous traiterons donc des situations les plus courantes : quel(s) type(s) de graphique(s) produire lorsque l'on dispose d'une, deux ou trois variables ? Quel(s) type(s) de graphique(s) produire lorsque les variables sont toutes numériques, toutes catégorielles, ou lorsqu'on dispose de variables des deux types ?
Pour chaque situation, un ou des exemples seront fournis à partir des données du tableau `penguins`. Cela sera

aussi l'occasion de présenter quelques subtilités liées à l'utilisation du package `ggplot2`.

3.5 Une seule variable numérique

Lorsque l'on souhaite examiner une unique variable numérique, deux types de représentations graphiques sont en général possibles :

1. les histogrammes : la variable d'intérêt est placée sur l'axe des `x` du graphique. Les valeurs utilisées sur l'axe des `y` est calculée automatiquement par le logiciel.
2. les nuages de points : la variable d'intérêt est placée sur l'axe des `y`. L'axe des `x` porte soit un simple numéro d'indice pour chaque observation, soit une unique valeur sans importance, la même pour toutes les observations.

Les syntaxes et options pour ces 2 types de graphiques sont présentées ci-dessous.

3.5.1 Les histogrammes

3.5.1.1 Syntaxe élémentaire

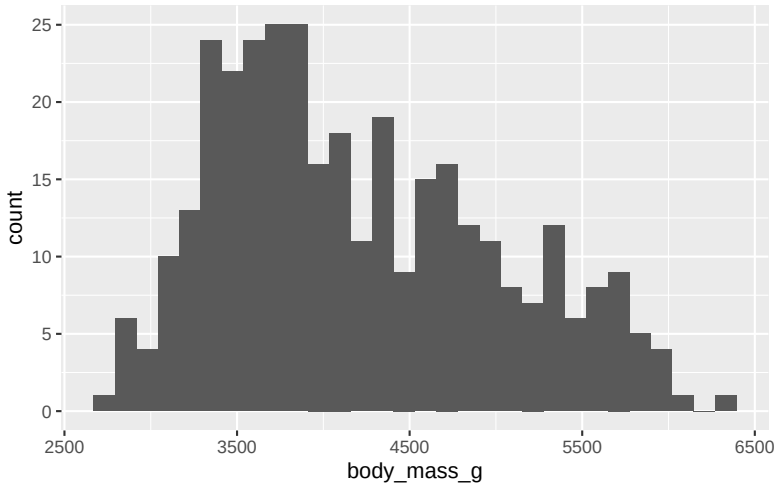
Imaginons que l'on s'intéresse à la variable `body_mass_g` du jeu de données `penguins`.

La syntaxe permettant de produire un histogramme, sous sa forme la plus simple, est la suivante :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Deux messages nous sont adressés par le logiciel :

1. `Message d'avis: Removed 2 rows containing non-finite values (stat_bin)`. Ce message indique, comme pour le premier nuage de points, que 2 individus du tableau `penguins` ont une masse corporelle inconnue (`NA`). Ces 2 individus (donc les deux lignes correspondantes), ont été ignorés pour produire ce graphique
2. `'stat_bin()' using 'bins = 30'. Pick better value with 'binwidth'`. Ce message indique que `R` a choisi pour nous les limites des classes utilisées pour faire l'histogramme. Sur un histogramme, la variable d'intérêt (toujours numérique et continue), qui apparaît sur l'axe des abscisses, est en effet "découpée" en plusieurs classes, en général de même taille, afin de permettre une représentation de la **distribution des valeurs**. Ici, `R` indique qu'il a créé 30 catégories pour nous, et que nous pouvons faire un choix différent grâce à l'argument `binwidth`. Nous y reviendrons un peu plus loin.

Sur ce graphique, l'axe des abscisses porte donc la variable continue "découpée" en classes de mêmes largeur, et l'axe des ordonnées renseigne sur le nombre (`count` ou fréquence absolue) d'individus observés dans chaque classe. Les zones du graphique où les barres sont les plus hautes indiquent donc les caractéristiques des individus observés le plus fréquemment. À l'inverse, les barres les plus courtes correspondent à des valeurs de masse rarement observées. Au final, ce type

de graphique permet de visualiser **la distribution des données pour une variable numérique continue**.

Ici, on constate qu'une majorité d'individus semble avoir des masses proches de 3500 grammes. Une autre portion non négligeable des individus (mais moins importante) semble avoir une masse légèrement supérieure à 4500 grammes. Enfin, les masses supérieures à 6000 grammes sont très rares. L'histogramme nous permet également de visualiser l'étendue des données : les manchots étudiés ici ont des masses qui s'étalent d'un peu plus de 2500 grammes à un peu moins de 6500 grammes.

3.5.1.2 Couleur

Pour rendre ce graphique plus facilement lisible, on peut en modifier la couleur :

- la couleur de remplissage des barres peut-être spécifiée grâce à l'argument `fill =`
- la couleur de contour des barres peut-être spécifiée grâce à l'argument `color =`

Une liste des couleurs disponibles dans `R` peut être affichée dans la console en tapant :

```
colors()
```

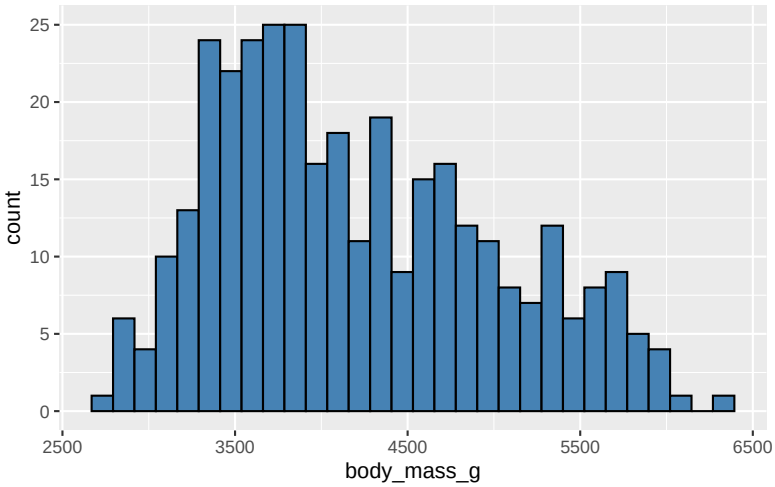
Vous pouvez voir à quelle couleur correspond chacun de ces noms dans ce document pdf.

Mettons à jour notre histogramme en ajoutant un peu de couleur :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black")
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Les 30 classes de masses sont maintenant plus facilement visibles et distingables.

3.5.1.3 À l'intérieur ou à l'extérieur de `aes()` ?

Les couleurs de remplissage et de contour des barres d'un histogramme font partie des caractéristiques esthétiques du graphique. Pourtant, elles ne sont pas précisées à l'intérieur de la fonction `aes()`. La raison est simple mais importante :

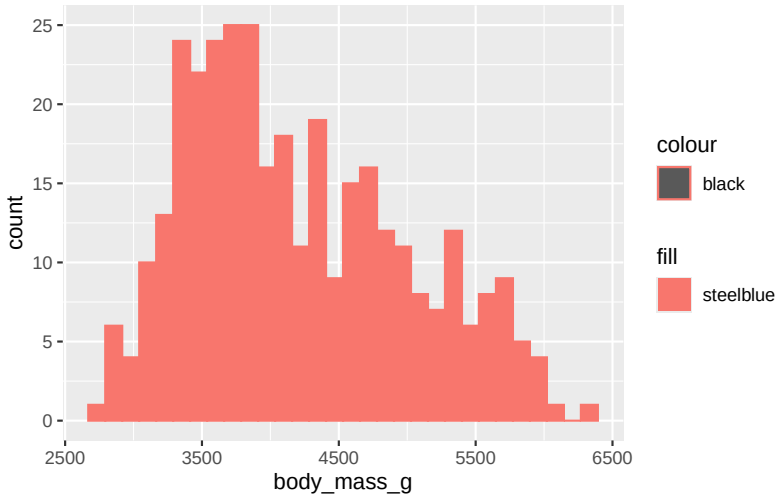
> **!** Important
>
> On place à l'intérieur de `aes()` uniquement les caractéristiques esthétiques du graphique que l'on souhaite associer à des **variables du jeu de données**.

Ici, les couleurs que l'on indique sont des constantes : toutes les barres ont les mêmes couleur de remplissage et de contour. On n'associe pas une variable du jeu de données à ces caractéristiques esthétiques. On place donc `fill =` et `color =` à l'extérieur de `aes()`. Si on se trompe, voilà ce qui se produit :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(aes(fill = "steelblue", color = "black"))
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Les couleurs qui apparaissent ne correspondent pas à ce qui est demandé, et une légende ne correspondant à rien apparaît à droite du graphique. La syntaxe utilisée ici suppose en effet que `"steelblue"` et `"black"` seraient des variables du jeu de données `penguins`. Puisqu'elles n'existent pas, `R` essaie de se débrouiller pour interpréter comme il peut ce qu'on lui demande, et finit par produire ce graphique incohérent. La couleur utilisée est la première couleur de la palette par défaut de `ggplot2`.

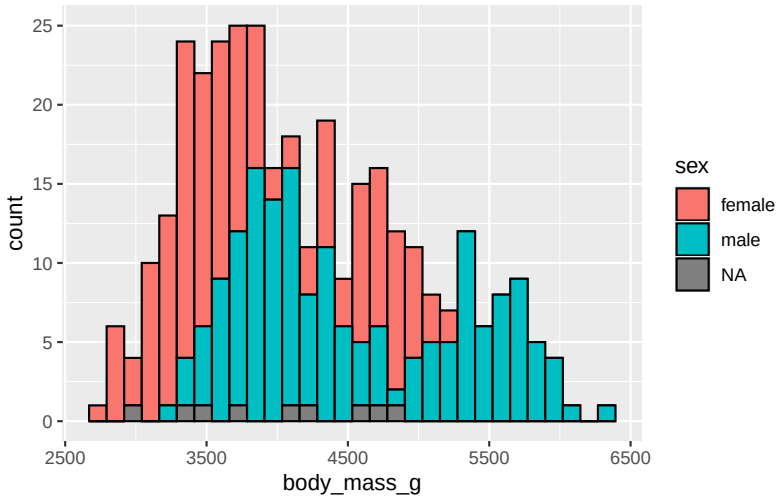
Pour élaborer des graphiques plus avancés, il faudra donc toujours vous poser la question suivante : la caractéristique esthétique que je souhaite modifier doit-elle être associée à une **valeur constante** que je fixe pour toutes les barres ou tous les points d'un graphique, et alors, je l'indique **en dehors** de `aes()`, ou est-elle au contraire associée à **une variable du jeu de données**, et alors, je l'indique **à l'intérieur** de `aes()`.

Il est bien sûr possible d'avoir un mélange des deux. Par exemple, le code suivant permet d'associer la couleur de remplissage au sexe des individus étudiés (variable `sex` du jeu de données `penguins`), et de spécifier une valeur constante pour la couleur de contour des barres (ici, le noir) :

```r
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(aes(fill = sex), color = "black")
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



On constate que toutes les barres ont un contour noir, mais que plusieurs couleurs de remplissage apparaissent maintenant, selon le sexe des individus, dans chaque classe de masse. Une légende adaptée est aussi créée automatiquement à droite du graphique. On apprend ainsi que les individus les plus lourds sont tous des mâles. On constate également que le sexe de certains individus est inconnu.

Au final, nous ne sommes déjà plus dans la situation où on examine une unique variable numérique. Nous avons en effet ici un graphique nous permettant de mettre en relation une variable numérique (la masse des individus en grammes) et une variable catégorielle (le sexe des individus). Nous reviendrons plus tard sur ce type de graphiques.

3.5.1.4 La largeur des classes

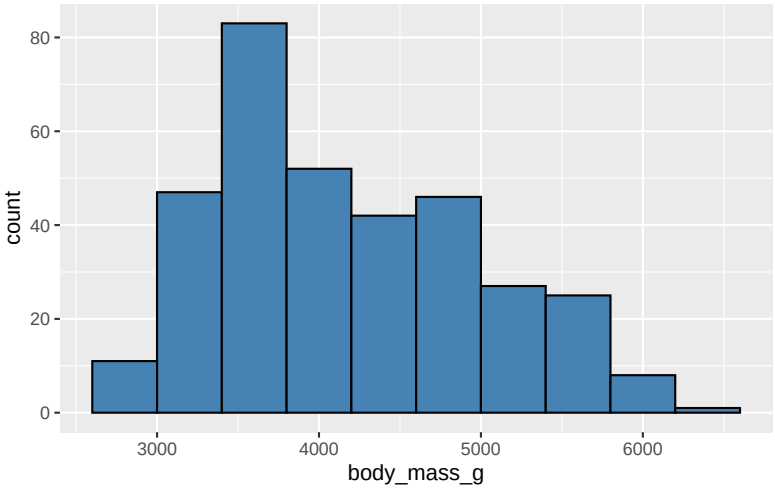
Comme évoqué plus haut, par défaut, R choisit arbitrairement de découper la variable numérique utilisée en 30 classes

de même largeur afin de produire l'histogramme. Ça n'est que rarement un bon choix, et malheureusement, il n'y a pas de règle permettant de définir à coup sûr le bon nombre de classes pour visualiser au mieux la distribution d'une variable numérique. Il faut en effet presque toujours procéder par essais-erreurs successifs. Il est possible d'ajuster les caractéristiques (nombre et/ou largeur) des classes de l'histogramme de l'une des 3 façons suivantes :

1. En ajustant le nombre de classes avec `bins`.
2. En précisant la largeur des classes avec `binwidth`.
3. En fournissant manuellement les limites des classes avec `breaks`.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 bins = 10)
```

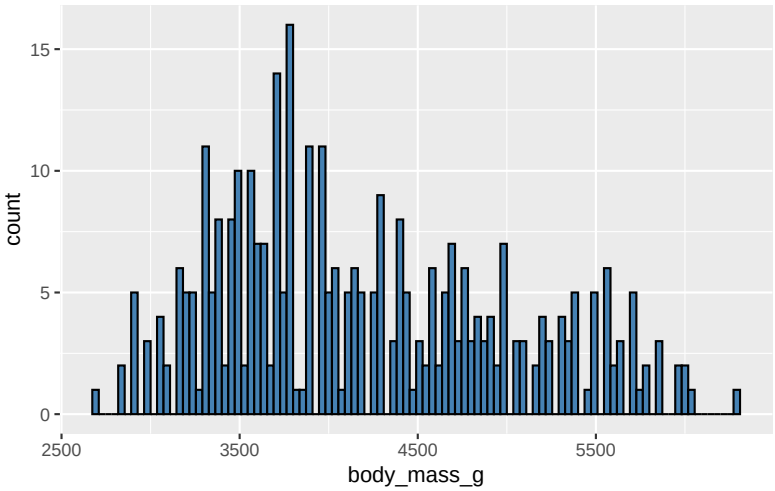
```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Ici, diminuer le nombre de classes à 10 a pour effet de trop lisser la distribution des données. On ne visualise plus les variations subtiles de la distribution. À l'inverse, trop augmenter le nombre de classes n'est pas pertinent non plus :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 bins = 100)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

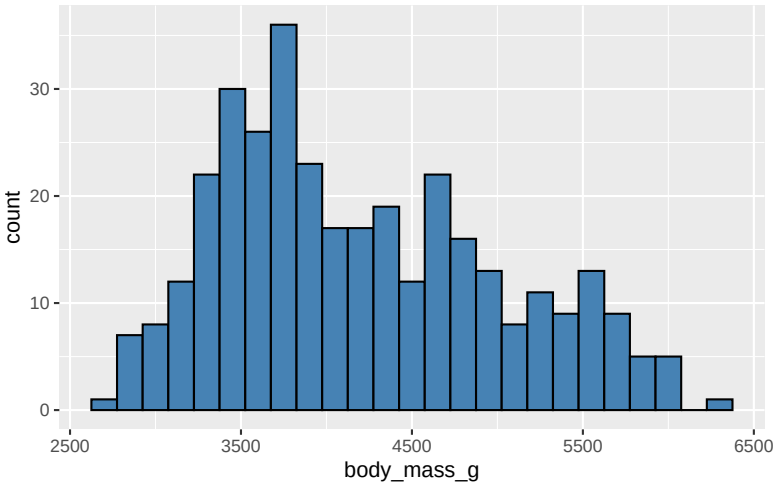


Ici, passer à 100 classes de taille génère un histogramme plein de trous, avec des classes très étroites, dont certaines sont très représentées, et immédiatement suivies ou précédées par des classes très peu représentées. Cela n'a pas de logique, et c'est presque toujours le signe qu'il faut réduire le nombre de classes.

Au final, pour ces données, un nombre de classes compris entre 20 et 30 semble un bon choix :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 bins = 25)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

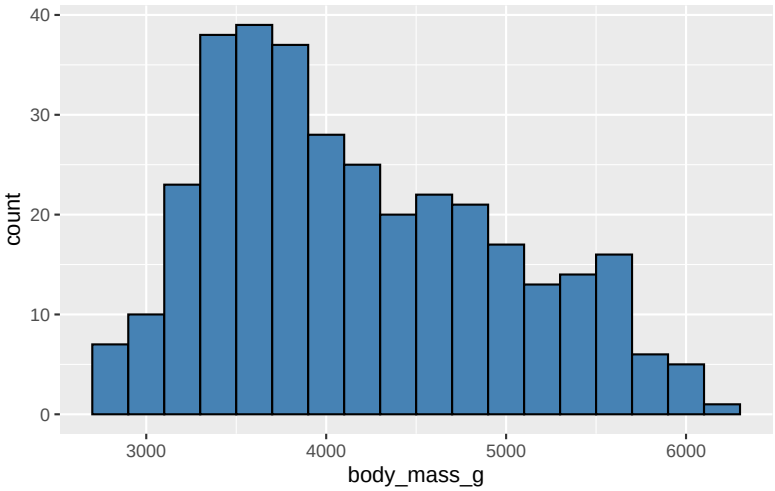


C'est un bon choix, entre trop peu d'information, et trop de bruit visuel. Évidemment, ce nombre sera différent pour chaque jeu de données. On constate ici à peu près 3 pics (autour de 3500 grammes, un peu au-dessus de 4500 grammes, et autour de 5500 grammes) qui reflètent bien la distribution de ces données.

On peut également modifier **la largeur des classes** (et non plus leur nombre) avec `binwidth` :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 binwidth = 200)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

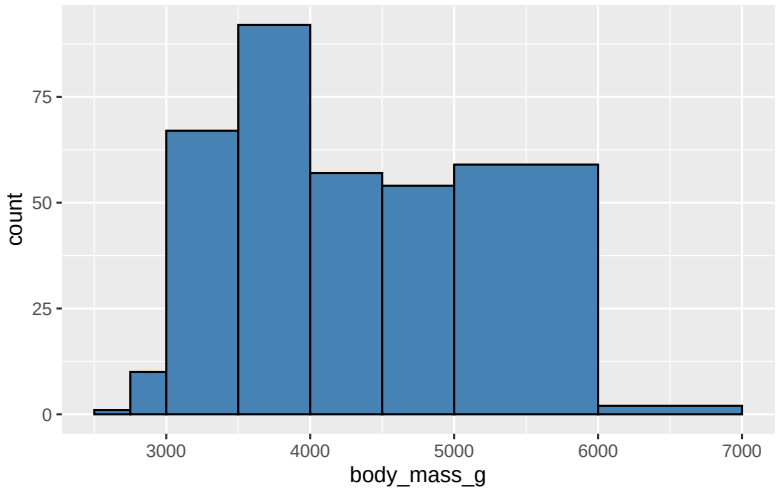


Ici, chaque catégorie recouvre 200 grammes. Avec l'argument **bins**, on indique à R combien on souhaite obtenir de classes, et il détermine automatiquement leur largeur. Avec **binwidth**, on indique la largeur des classes souhaitées, et R détermine automatiquement le nombre de classes nécessaires pour couvrir la totalité des données.

Enfin, il est possible de déterminer manuellement les limites des classes souhaitées avec l'argument **breaks** :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 breaks = c(2500, 2750, 3000, 3500, 4000, 4500, 5000, 6000, 7000))
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Vous constatez ici que les choix effectués ne sont pas très pertinents : toutes les classes n'ont pas la même largeur. Cela rend l'interprétation difficile. Il est donc vivement conseillé, pour spécifier `breaks`, de créer des suites régulières, comme avec la fonction `seq()` (consultez son fichier d'aide et les exemples) :

```
limites <- seq(from = 2500, to = 6500, by = 250)
limites
```

```
 [1] 2500 2750 3000 3250 3500 3750 4000 4250 4500 4750 5000 5250 5500 5750 6000
[16] 6250 6500
```

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 breaks = limites)
```

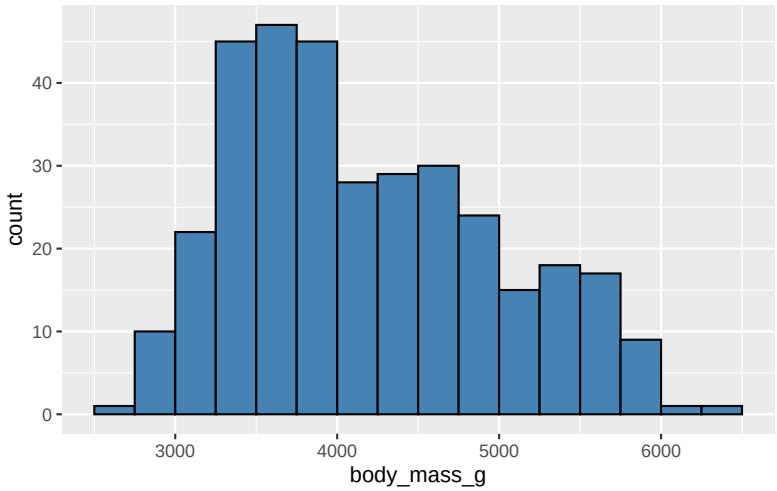


Figure 3.4: Un exemple d'utilisation de l'argument `breaks`

Il est important que toute la gamme des valeurs de `body_mass_g` soit bien couverte par les limites des classes que nous avons définies, sinon, certaines valeurs sont omises et l'histogramme est donc incomplet/incorrect. Une façon de s'en assurer est d'afficher le résumé des données pour la colonne `body_mass_g` du jeu de données `penguins` :

```
summary(penguins$body_mass_g)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   2700    3550    4050    4202    4750    6300       2
```

On voit ici que les masses varient de 2700 à 6300 grammes. Les classes que nous avons définies couvrent une plage de masses plus large (de 2500 à 6500). Toutes les données sont donc bien intégrées à l'histogramme.

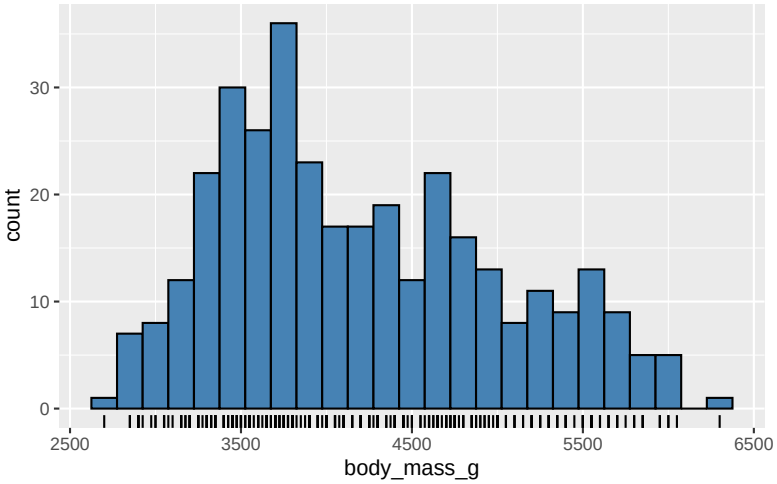
3.5.1.5 `geom_rug` et `geom_density`

La fonction `geom_histogram()` n'est pas la seule qui permette de visualiser la distribution des données. Il est en effet possible d'utiliser d'autres objets géométriques, en plus ou à la place de `geom_histogram()` pour ajouter de l'information sur le graphique, ou pour visualiser différemment la distribution des mêmes données.

La fonction `geom_rug()` permet d'ajouter les données réelles sous forme de segments, sous un histogramme. Cela prend souvent l'aspect d'une sorte de tapis, d'où le nom de la fonction ("rug" signifie "tapis" en anglais). Pour ajouter une couche supplémentaire au graphique, on ajoute simplement un `+` à la fin de la dernière ligne, et sur la ligne suivante, on ajoute un objet géométrique supplémentaire :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
                 bins = 25) +
  geom_rug()
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

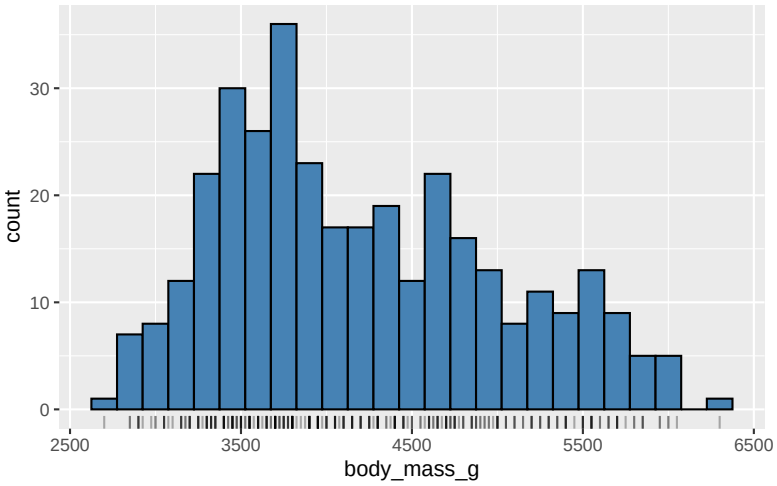


Les tirets qui sont maintenant visibles en-dessous de l'histogramme correspondent aux 342 valeurs de masses réellement observées dans le jeu de données. Puisque certaines tailles ont été observées plusieurs fois, faire des tirets semi-transparents nous permettra de mieux visualiser quelles tailles ont été observées fréquemment ou rarement. On peut régler la transparence des éléments d'un graphique avec l'argument `alpha =`, qui prend des valeurs comprises entre 0 (transparence totale) et 1 (opacité totale) :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(fill = "steelblue", color = "black",
```

```
                   bins = 25) +
  geom_rug(alpha = 0.3)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

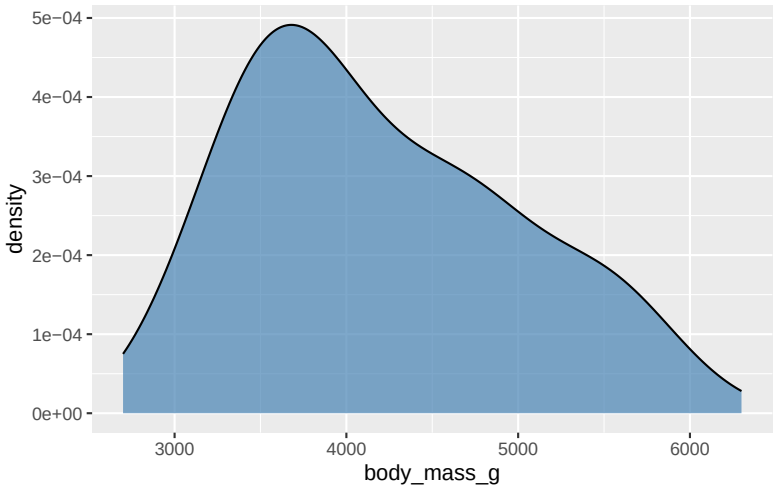


Les tirets sont maintenant d'autant plus foncés que les tailles ont été observées un grand nombre de fois. On retrouve bien ici la distribution décrite plus haut, avec 3 principaux groupes de valeurs. Cela révèle certainement en partie la complexité des données : ces tailles correspondent en effet aux mesures effectuées chez 3 espèces distinctes qui peuvent avoir des caractéristiques différentes, sans compter que le sexe des individus, qui n'apparaît pas ici, entre aussi probablement en jeu. Nous y reviendrons plus tard.

La fonction `geom_density()` permet de s'affranchir de la question du nombre ou de la largeur des classes de taille :

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_density(fill = "steelblue", color = "black", alpha = 0.7, bw = 300)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```



On obtient une sorte d'histogramme lissé qui fait bien apparaître les 3 tailles les plus fréquentes (au niveau des 3 "bosses" du graphique). Inutile ici de spécifier un nombre de classes de taille, ou leur largeur : le lissage est ici automatique. On peut modifier l'importance du lissage avec l'argument `bw`, mais la valeur choisie par défaut par `R` est généralement tout à fait satisfaisante. Vous pouvez essayer avec une valeur de lissage de 30, puis de 500 pour vous rendre compte de l'effet de ce paramètre.

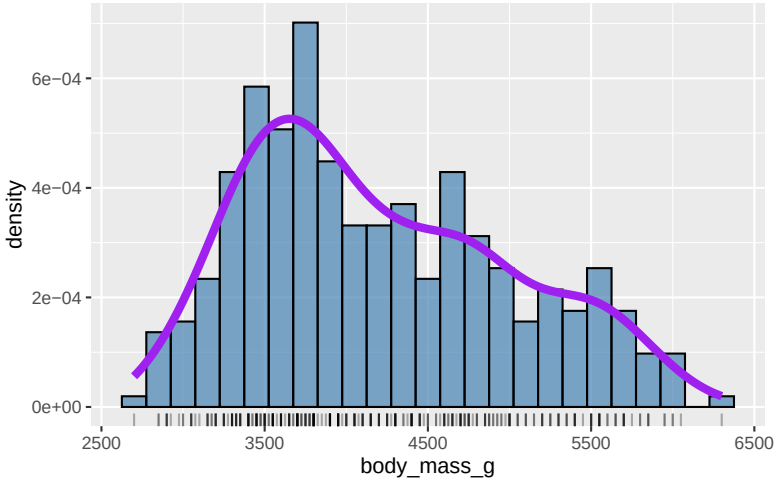
Notez également que si l'histogramme présentait des valeurs d'abondance sur l'axe des `y` (des nombres d'individus), le graphique de densité présente, comme son nom l'indique, l'information de densité des observations. Cela signifie que la surface totale sous la courbe (en bleu) vaut 1. Cela peut s'avérer utile pour comparer plusieurs distributions pour lesquelles on disposes de tailles d'échantillons très différentes.

Enfin, on peut créer un graphique qui présentera à la fois l'histogramme (avec `geom_histogram()`), les données individuelles (avec `geom_rug()`) et la courbe de densité (avec `geom_density()`). Mais pour que tout s'affiche correctement, il faut indiquer à `geom_histogram` que l'axe des `y` doit porter les densités et non les abondances. On fait cela en précisant `y = after_stat(density)`, cela indique à `R` que la variable `density` ne figure pas dans le tableau `penguins`, mais qu'elle est calculée par la fonction `geom_histogram()` :

```r
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(aes(y = after_stat(density)),
                 fill = "steelblue", color = "black",
                 bins = 25, alpha = 0.7) +
  geom_rug(alpha = 0.3) +
  geom_density(color = "purple", linewidth = 2)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```



Notez l'utilisation des arguments `alpha`, `color` et `size`, pour modifier l'aspect de différents éléments du graphique. Assurez-vous d'avoir compris comment on les utilise, et faites vos propres expériences.

3.5.1.6 Un mot sur la position de la fonction `aes()`

Sur le dernier exemple, vous constatez que la fonction `aes()` apparaît une fois à l'intérieur de la fonction `ggplot()`, et une autre fois à l'intérieur de `geom_histogram()`. Pourquoi ne pas avoir tapé, plus simplement :

```
ggplot(penguins, aes(x = body_mass_g, y = after_stat(density))) +
  geom_histogram(fill = "steelblue", color = "black",
                 bins = 25, alpha = 0.7) +
  geom_rug(alpha = 0.3) +
  geom_density(color = "purple", linewidth = 2)
```

L'explication est relativement simple, mais importante :

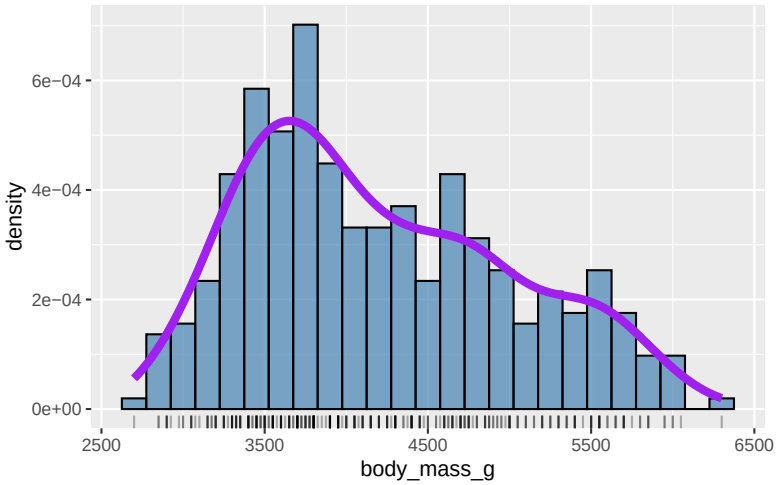
> **!** Important
>
> Ce qui est spécifié dans la fonction `ggplot()` s'applique à toutes les couches du graphiques (donc ici, aux 3 couches `geom_histogram()`, `geom_rug()` et `geom_density()`).
> Ce qui est spécifié dans une fonction `geom_...()` ne s'applique qu'à cette couche géométrique particulière.

Ainsi, ajouter `y = after_stat(density)` à l'intérieur de `ggplot()` renvoie donc un message d'erreur, car seule la fonction `geom_histogram()` calcule la variable `density`, seule la fonction `geom_histogram()` sait quoi faire de cette variable. Dans notre exemple, il est en revanche logique d'ajouter `aes(x = body_mass_g)` dans la fonction `ggplot()`, car nos trois couches géométriques ont besoin de cet argument, et pour les 3 couches géométriques, on associe bien cette variable `body_mass_g` à l'axe des `x`. Toutefois, rien ne nous empêche d'écrire ceci à la place :

```
ggplot(data = penguins) +
  geom_histogram(aes(x = body_mass_g, y = after_stat(density)),
                 fill = "steelblue", color = "black",
                 bins = 25, alpha = 0.7) +
  geom_rug(aes(x = body_mass_g),
           alpha = 0.3) +
  geom_density(aes(x = body_mass_g),
               color = "purple", linewidth = 2)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_density()`).
```



C'est plus long, mais c'est tout à fait correct et ça produit exactement le même résultat qu'auparavant.

3.5.2 Les nuages de points et stripcharts

Pour ces deux types de graphiques, la variable numérique sera portée par l'axe des `y`, et toutes les valeurs seront visibles, de façon non agrégée (contrairement aux histogrammes où les valeurs individuelles sont rassemblées à l'intérieur de classes). La différence entre les deux types de graphiques tient à la nature des informations qui figureront sur l'axe des `x` :

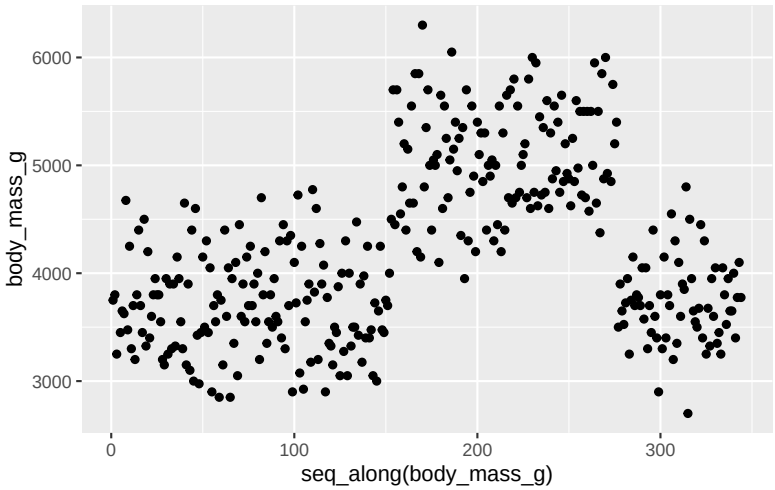
- Pour les nuages de points, l'axe des `x` portera simplement l'information du numéro d'observation pour chaque individu. L'individu placé sur la première ligne du tableau de données portera l'indice `1`. L'individu placé sur la deuxième ligne du tableau de données portera l'indice `2`, et ainsi de suite jusqu'à l'individu placé sur la dernière ligne du tableau (il portera ici l'indice 344 puisque le tableau compte 344 lignes)
- Pour un stripchart, l'axe des `x` portera une unique valeur, la même pour tous les individus

Dans les deux cas, l'axe des `x` ne nous sera pas vraiment utile. Il nous servira simplement à afficher des points sur un graphique, mais puisque nous ne disposons que d'une unique variable, c'est bien l'axe des `y` qui nous intéressera en priorité. Pour faire un nuage de points, on utilise `geom_point()`, et

pour un stripchart `geom_jitter()`. Commençons par examiner le nuage de points pour la variable `body-mass-g` :

```
ggplot(penguins, aes(x = seq_along(body_mass_g), y = body_mass_g)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



C'est la fonction `seq_along()`, que l'on associe à l'axe des `x`, qui permet de faire apparaître les numéros de lignes du tableau `penguins`. On constate ici que 3 groupes de points sont présents :

1. Pour les lignes 1 à 150 (environ), un premier groupe de points présente des masses comprises entre 3000 et 4800 grammes environ.
2. Pour les lignes 151 à 275 (environ), un second groupes de points présente des masses comprises entre 4000 et plus de 6000 grammes.
3. Pour les lignes 276 à 344 (environ), un troisième groupe de points présente des valeurs similaires à celles du premier groupe.

En examinant le tableau `penguins` de plus près, on se rend compte que les 3 espèces de manchots sont présentées dans l'ordre. Ainsi, ces 3 groupes correspondent à 3 espèces différentes. Pour le visualiser, il suffit d'associer la variable `species` à la couleur des points. Puisqu'on cherche à associer

une variable du tableau de données à une caractéristique esthétique d'un objet géométrique, on renseigne `color = species` à l'intérieur de `aes()` :

```
ggplot(penguins, aes(x = seq_along(body_mass_g), y = body_mass_g)) +
  geom_point(aes(color = species))
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

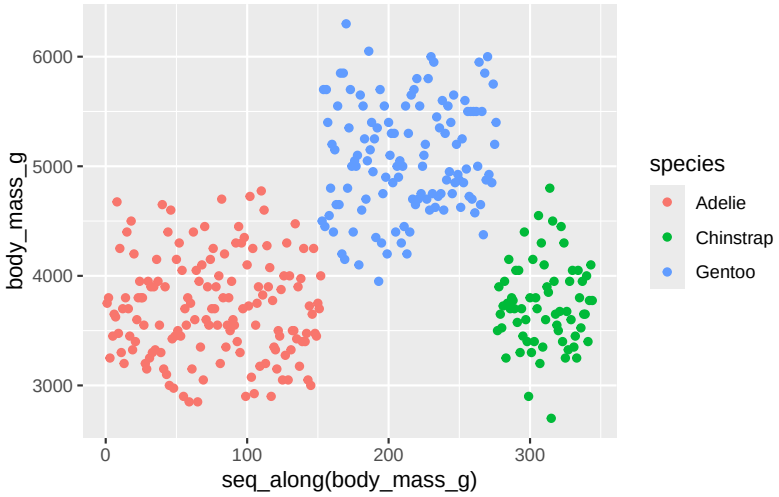


Figure 3.5: Nuage de points des masses corporelles des 3 espèces de manchots

Attention, nous ne sommes déjà plus dans la situation d'une unique variable numérique : nous avons ici visualisé 2 variables : une numérique (portée par l'axe des `y`) et une catégorielle (l'espèce représentée par la couleur des points). Ici, on constate que les espèces Adélie et Chinstrap semblent avoir approximativement la même gamme de masses, alors que les Gentoo semblent nettement plus lourds.

Comme pour les histogrammes, on peut utiliser des caractéristiques esthétiques variées pour modifier l'apparence des points :

- `alpha` : la transparence. Choisir une valeur comprise entre 0 (invisible) et 1 (totalement opaque)
- `size` : la taille des points

- `color` : la couleur des points (ou de leur contour pour les symboles qui permettent de spécifier une couleur de remplissage et une couleur de contour)
- `fill` : la couleur de remplissage des points (pour les symboles qui permettent de spécifier une couleur de remplissage et une couleur de contour)
- `shape` : pour modifier les symboles utilisés. Les symboles possibles sont codés ainsi :

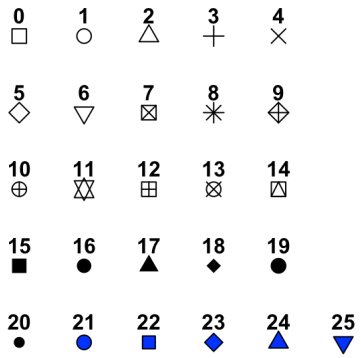
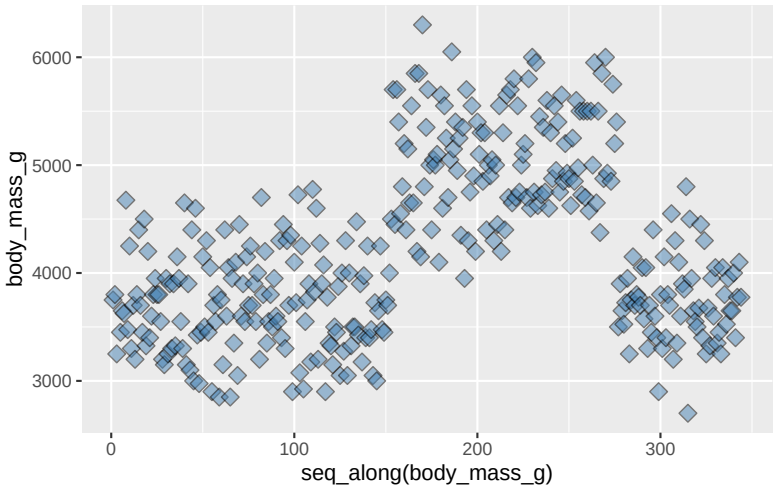


Figure 3.6: Liste des symboles et codes correspondants pour les graphiques faisant apparaître des points. Pour les symboles 21 à 25, il sera possible de spécifier une couleur de remplissage `fill` et une couleur de contour `color`. Pour tous les autres symboles, les changements de couleurs se feront avec l'argument `color`.

Chacune de ces caractéristiques esthétiques peut être associée à une variable d'un tableau (il faut alors le spécifier à l'intérieur de `aes()`), ou à une valeur unique, constante et identique pour tous les points du graphique (il faut alors le spécifier à l'extérieur de `aes()`). Par exemple :

```
ggplot(penguins, aes(x = seq_along(body_mass_g), y = body_mass_g)) +
  geom_point(shape = 23, fill = "steelblue", color = "black",
             size = 3, alpha = 0.5)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

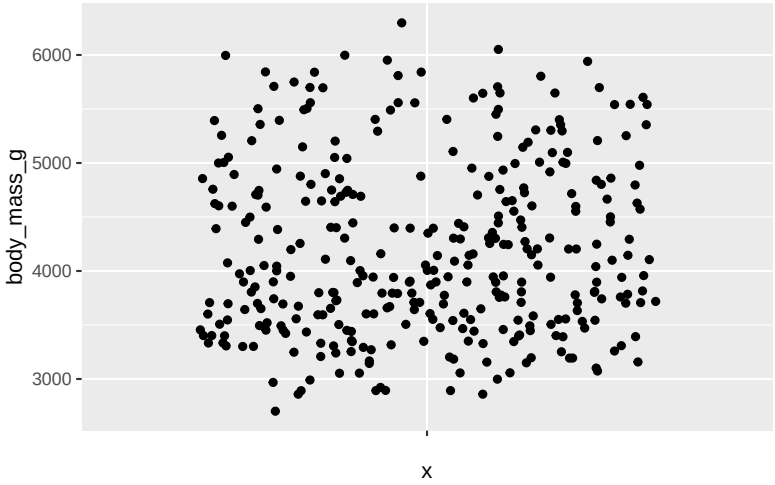


L'ajout de la transparence permet de régler le problème des points qui se superposent (un phénomène nommé "overplotting").

Examinons à présent un exemple de stripchart :

```
ggplot(penguins, aes(x = "", y = body_mass_g)) +
  geom_jitter()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



En indiquant `x = ""`, nous créons une unique catégorie pour l'axe des abscisses, qui sera utilisée pour placer les valeurs de tous les individus. Les valeurs de `body_mass_g` sont lues sur

l'axe des `y`, comme pour un nuage de point classique. Si les points apparaissent dispersés, c'est en raison de 2 arguments spécifiques de la fonction `geom_jitter()` :

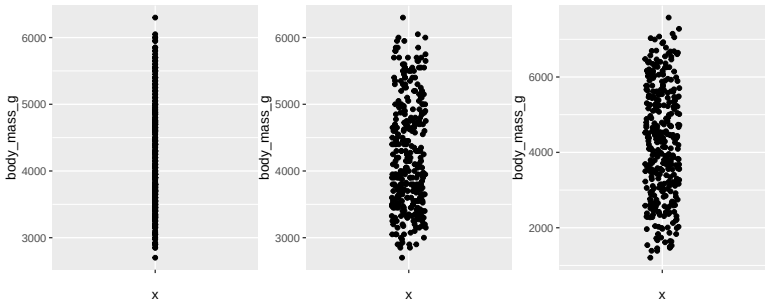
- `width =` permet de spécifier l'étendue horizontale du bruit aléatoire qui sera utilisé pour placer les points
- `height =` permet de spécifier l'étendue verticale du bruit aléatoire qui sera utilisé pour placer les points

Si nous ne renseignons pas nous même ces deux arguments, ils sont fixés automatiquement par le logiciel, ce qui n'est pas souhaitable, notamment pour le bruit vertical. Pour mieux comprendre, voyons ce qui se passe dans 3 situations :

```
ggplot(penguins, aes(x = "", y = body_mass_g)) +
  geom_jitter(width = 0, height = 0)

ggplot(penguins, aes(x = "", y = body_mass_g)) +
  geom_jitter(width = 0.1, height = 0)

ggplot(penguins, aes(x = "", y = body_mass_g)) +
  geom_jitter(width = 0.1, height = 2000)
```



(a) Pas de dispersion horizontale, pas de dispersion verticale

(b) Faible dispersion horizontale, pas de dispersion verticale

(c) Faible dispersion horizontale, forte dispersion verticale

Figure 3.7: Trois exemples de stripchart

- Le premier exemple (Figure 3.7a) ne présente aucune dispersion, ni horizontale (`width = 0`), ni verticale (`height = 0`). Les points apparaissent donc tous alignés, ils ont en effet tous la même valeur sur l'axe des abscisses. Leur position sur l'axe des `y` reflète la masse réellement observée pour chaque individu.

Cette façon de représenter les données n'est pas très utile car la superposition des points vient empêcher la visualisation correcte de la distribution : ici, il est impossible de dire quelles sont les masses les plus fréquemment observées ou les plus rarement observées.

- Le second exemple (Figure 3.7b) présente une dispersion horizontale modérée (`width = 0.1`) et pas de dispersion verticale (`height = 0`). Ici, tous les points ne sont plus alignés sur une seule droite. Puisque nous avons fixé `width = 0.1`, la position horizontale des points est choisie aléatoirement par `R` : il ajoute un léger bruit horizontal aléatoire, soit positif, soit négatif, avant de placer les points le long de l'axe des abscisses. Plus la valeur de `width` sera élevée, plus l'étendue du bruit horizontal sera importante. Sur l'axe des `y` en revanche, aucun bruit n'a été ajouté (`height = 0`). La position des points le long de cet axe reflète donc parfaitement la masse de chaque individu telle qu'enregistrée dans le tableau `penguins` et c'est bien ce que nous voulons. D'ailleurs, on constate que l'axe des ordonnées est strictement identique (même étendue, même graduations…) pour les 2 premiers sous-graphiques. C'est ce type de représentation que nous recherchons. En effet, l'absence de bruit vertical nous permet de visualiser correctement (donc sans distorsion) la variable numérique choisie (ici `body_mass_g`), et le bruit horizontal nous permet d'étaler légèrement les points de part et d'autres d'un axe vertical virtuel, ce qui a pour effet de réduire l'overplotting, et ce qui nous permet donc de visualiser les zones où les points sont plus nombreux/denses et les zones où les observations sont plus rares. Ici, on observe une majorité de points entre 3000 et 4000 grammes, une densité de points intermédiaire entre 4000 et 5000 grammes, et des points moins nombreux (donc moins d'individus) pour les masses supérieures à 5000 grammes.
- Le troisième exemple (Figure 3.7c) présente une dispersion horizontale modérée (`width = 0.1`) et une importante dispersion verticale (`height = 2000`). Cela signifie que la position des points sur l'axe des `y` ne reflète plus les vraies valeurs de masses enregistrées dans le ta-

bleau `penguins`, mais des valeurs de masses auxquelles un bruit aléatoire a été ajouté ou retiré. C'est ce qui explique que l'axe des ordonnées ne présente pas la même échelle que pour les 2 autres graphiques. Ce n'est évidemment pas souhaitable, car si nous voulons bel et bien ajouter un bruit horizontal pour éviter la superposition des points, il est essentiel de ne pas modifier la position verticale des points qui nous renseigne sur la variable d'intérêt. Ici, la Figure 3.7c présente un axe des `y` différent des 2 autres sous-figures, et la position verticale des points a été tellement altérée qu'on ne peut plus distinguer la sur-abondance de données entre 3000 et 4000 grammes, ni la sous-représentation des observations au-dessus de 5000 grammes. Il sera donc important à l'avenir de toujours fixer `height = 0` pour faire un stripchart correct.

> **!** Important
>
> Sur un stripchart :
>
> - la position **verticale** des points ne doit jamais être modifiée. On fixera donc toujours `height = 0`
> - la position **horizontale** des points doit être modifiée afin d'éviter l'overplotting et de visualiser les zones de fortes et faibles densités de points. On choisira donc en général des valeurs de `width` comprises entre `0.1` et `0.4`

Enfin, puisqu'un stripchart permet d'afficher des points sur un graphiques, les arguments permettant de modifier l'aspect des points sont les mêmes que pour les nuages de points. Par exemple :

```
ggplot(penguins, aes(x = "", y = body_mass_g)) +
  geom_jitter(aes(color = species, shape = species),
              size = 3, alpha = 0.6,
              width = 0.1, height = 0)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

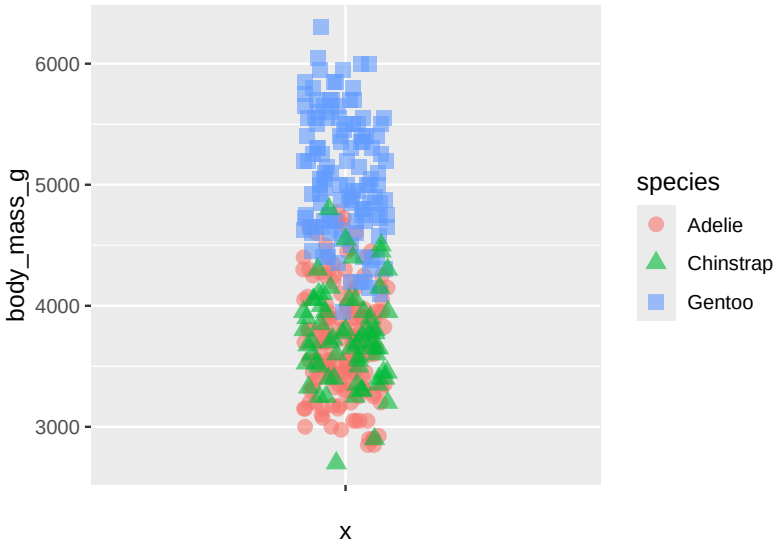


Figure 3.8: Un exemple de stripchart

Sur cette figure, comme pour le nuage de points réalisé plus haut, j'ai associé la variable `species` à la couleur des points (donc à l'intérieur de `aes()`). J'ai également associé cette variable à la forme des points `shape = species` à l'intérieur de `aes()`. C'est ce qui explique que chacune des 3 espèces apparaît sous la forme de symboles de formes et de couleurs différents. Pour limiter l'overplotting, j'ai spécifié un bruit horizontal, et j'ai fixé le bruit vertical à zéro. Enfin j'ai augmenté la taille des symboles (avec `size = 3`, en dehors de `aes()` car `3` est une constante qui s'appliquera à tous les points du graphique de la même manière) et leur transparence (avec `alpha = 0.6`, toujours en dehors de `aes()` pour la même raison). On constate ici encore que les masses corporelles des manchots Adélie et Chinstrap sont très similaires, et inférieures à celles de l'espèce Gentoo.

3.5.3 Exercices

1. À quoi sert l'argument `stroke` pour les nuages de points et les stripcharts ?
2. Créez de nouveaux graphiques (histogramme et diagramme de densité) avec la variable contenant l'information de la longueur des nageoires des manchots `flipper_length_mm`. Décrivez les graphiques

obtenus. Vos observations sont-elles cohérentes avec ce que nous savons maintenant des masses individuelles ?

3. Visualisez ces données avec un nuage de points ou un stripchart. Retrouvez-vous les mêmes informations de distribution ?

3.6 Une seule variable catégorielle

3.6.1 Les diagrammes bâtons

Comme nous l'avons vu plus haut, les histogrammes permettent de visualiser la distribution d'une **variable numérique continue**. Souvent, on souhaite visualiser la distribution d'une **variable catégorielle**. C'est une tâche relativement aisée puisqu'elle consiste simplement à compter combien d'éléments tombent dans chacune des catégories (ou modalités) de la variable catégorielle. Le meilleur moyen de visualiser de telles données de comptage (*aka* fréquences) est de réaliser un diagramme bâtons, autrement appelé **barplot** ou **barchart**.

Une difficulté, toutefois, concerne la façon dont les données sont présentées : est-ce que la variable d'intérêt est "précomptée" ou non ? Par exemple, le code ci-dessous crée 2 `data.frame` qui représentent la même collection de fruits : 3 pommes, 2 oranges et 4 bananes :

```
panier <- tibble(
  fruit = c("pomme", "pomme", "banane", "pomme", "orange", "banane", "orange", "banane",
)

panier_counted <- tibble(
  fruit = c("pomme", "orange", "banane"),
  nombre = c(3, 2, 4)
)
```

Le tableau `panier` contient des données qui n'ont pas encore été comptées. Le tableau contient donc une unique variable nommée `fruit` :

```
panier
```

```
# A tibble: 9 x 1
  fruit
  <chr>
1 pomme
2 pomme
3 banane
4 pomme
5 orange
6 banane
7 orange
8 banane
9 banane
```

À l'inverse, le tableau `panier_counted` contient des données qui ont déjà été comptées. Le tableau contient donc 2 variables dans 2 colonnes distinctes : une colonne `fruit` et une colonne `nombre`, mais seulement 3 lignes puisque seulement 3 modalités (les catégories de la variable catégorielle) sont présentes pour la variable `fruit` :

```
panier_counted
```

```
# A tibble: 3 x 2
  fruit  nombre
  <chr>   <dbl>
1 pomme       3
2 orange      2
3 banane      4
```

Les deux tableaux `panier` et `panier_counted` représentent exactement les mêmes données, mais sous deux formats différents. Du fait de ces deux formats possibles, deux objets géométriques distincts devront être utilisés pour représenter les données. Le graphique obtenu sera le même, mais à chaque format de tableau son `geom_...()`.

3.6.1.1 `geom_bar()` et `geom_col()`

Pour visualiser les données non pré-comptées, on utilise `geom_bar()` :

```r
ggplot(panier, aes(x = fruit)) +
  geom_bar()
```

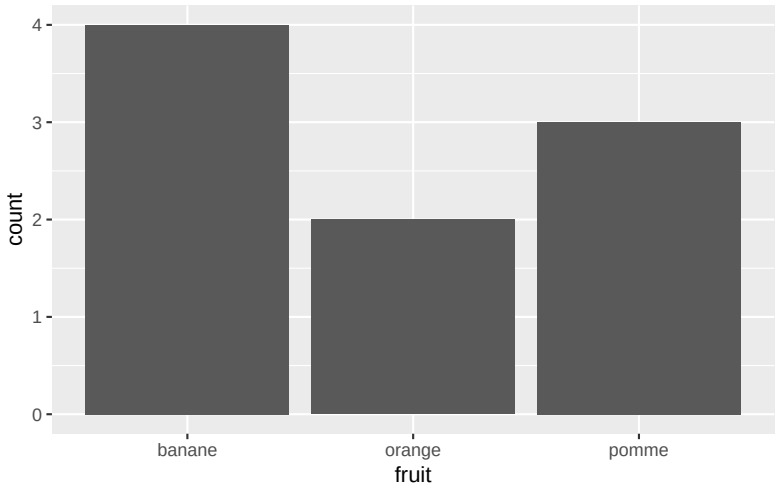


Figure 3.9: Barplot pour des données non pré-comptées.

Pour visualiser les données déjà pré-comptées, on utilise `geom_col()` :

```r
ggplot(panier_counted, aes(x = fruit, y = nombre)) +
  geom_col()
```

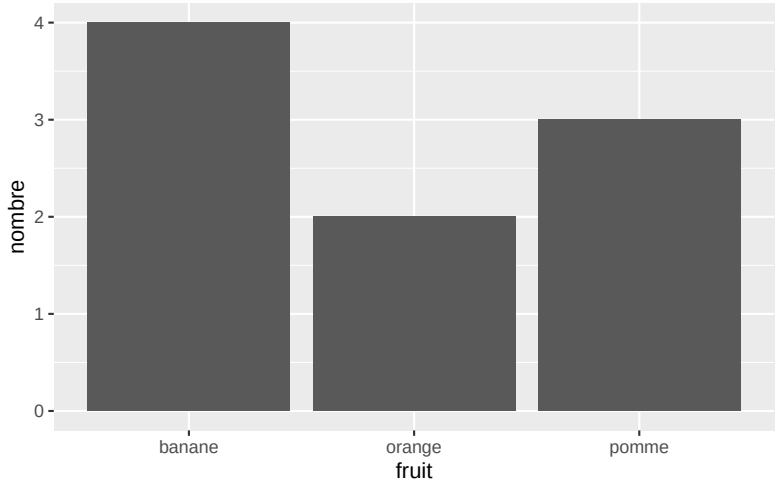


Figure 3.10: Barplot pour des données pré-comptées.

Notez que les figures Figure 3.9 et Figure 3.10 sont absolument identiques (à l'exception du titre de l'axe des ordonnées), mais qu'elles ont été créées à partir de 2 tableaux de données différents. En particulier, notez que :

- Le code qui génère la figure Figure 3.9 utilise le jeu de données `panier`, et n'associe pas de variable à l'axe des ordonnées : dans la fonction `aes()`, seule la variable associée à `x` est précisée. C'est la fonction `geom_bar()` qui calcule automatiquement les abondances (ou fréquences) pour chaque catégorie de la variable `fruit`. La variable `count` est ainsi générée automatiquement et associée à `y`.
- Le code qui génère la figure Figure 3.10 utilise le jeu de données `panier_counted`. Ici, c'est bien l'utilisateur qui associe la variable `nombre` à l'axe des `y` à l'intérieur de la fonction `aes()`. La fonction `geom_col()` a besoin de 2 variables (une variable catégorielle pour l'axe des `x` et une numérique pour l'axe des `y`) pour fonctionner.

Autrement dit, lorsque vous souhaiterez créer un diagramme bâtons, il faudra donc au préalable vérifier de quel type de données vous disposez pour choisir l'objet géométrique approprié :

> **!** Diagrammes bâtons
>
> - Si la variable catégorielle n'est pas pré-comptée dans le tableau de données → `geom_bar()`. La variable catégorielle est associée à l'esthétique `x` du graphique. On ne renseigne pas `y`.
> - Si la variable catégorielle est pré-comptée dans le tableau de données → `geom_col()`. La variable catégorielle est associée à l'esthétique `x` du graphique. On associe explicitement les comptages à l'esthétique `y` du graphique.

Enfin, notez que l'ordre des modalité (ou catégories) qui apparaissent sur l'axe des abscisses est l'ordre alphabétique : la modalité `banane` apparaît à gauche, puis la modalité `orange` et enfin la modalité `pomme`. Bien souvent, cet ordre alphabétique n'est pas pertinent. Nous verrons plus loin comment faire pour trier les catégories par ordre croissant ou décroissant. C'est en effet une possibilité intéressante qui est im-

possible pour les histogrammes (car l'axe des `x` porte une variable numérique continue qu'il est impossible de "mélanger"), mais souvent vivement recommandée pour les diagrammes bâtons.

3.6.1.2 Un exemple concret

Revenons aux manchots. Imaginons que nous souhaitions connaître le nombre d'individus étudiés pour chaque espèce. Dans le jeu de données `penguins`, la variable `species` indique à quelle espèce appartiennent chacun des 344 individus étudiés. Une façon simple de représenter ces données est donc la suivante :

```
ggplot(penguins, aes(x = species)) +
  geom_bar()
```

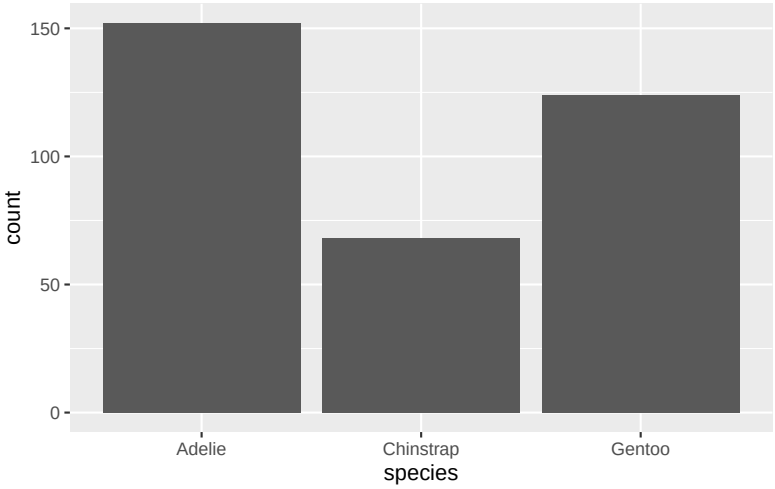


Figure 3.11: Effectifs pour les 3 espèces de manchots étudiées

Ici, `geom_bar()` a compté le nombre d'occurrences de chaque espèce dans le tableau `penguins` et a automatiquement associé ce nombre à l'axe des ordonnées.

Là encore, les modalités sont triées par ordre alphabétique sur l'axe des abscisses. Il est généralement plus utile de trier les catégories par ordre décroissant. Nous pouvons faire cela facilement grâce à la fonction `fct_infreq()` du package `forcats`, qui permet de modifier l'ordre des modalités d'une

variable catégorielle (ou facteur). Si vous avez installé le `tidyverse`, le package `forcast` doit être disponible sur votre ordinateur. N'oubliez pas de le charger si besoin :

```
library(forcats)
ggplot(penguins, aes(x = fct_infreq(species))) +
  geom_bar()
```

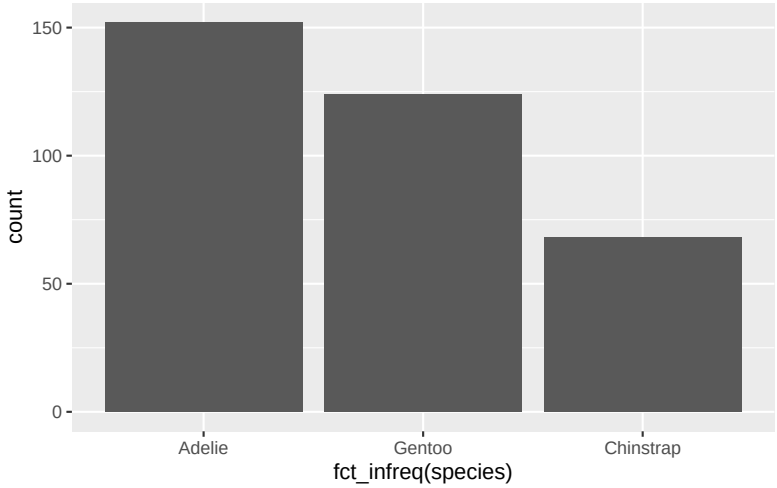


Figure 3.12: Effectifs pour les 3 espèces de manchots étudiées, triés en ordre décroissant

Ordonner les catégories par ordre décroissant est souvent indispensable afin de faciliter la lecture du graphique et les comparaisons entre catégories.

Si nous souhaitons connaître le nombre précis d'individus de chaque espèce, il nous faut faire appel à plusieurs fonctions du package `dplyr` que nous détaillerons dans le chapitre Chapitre 4. Ci-dessous, nous créons un nouveau tableau `species_table` contenant le nombre d'individus de chaque espèce et les espèces sont ordonnées par abondance décroissante :

```
species_table <- penguins |>   # On prend le tableau penguins, puis...
  count(species) |>            # On compte les effectifs de chaque espèce, puis...
  arrange(desc(n))              # On trie par effectif décroissants ...
species_table                   # Enfin, on affiche la nouvelle table
```

```
# A tibble: 3 x 2
  species       n
  <fct>     <int>
1 Adelie      152
2 Gentoo      124
3 Chinstrap    68
```

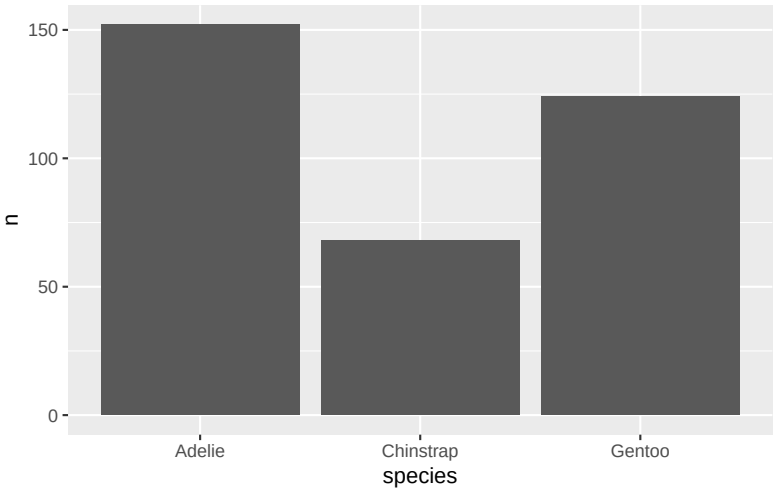
Ici, la table a été triée par effectifs décroissants. Mais attention, **les niveaux** du facteur `species` n'ont pas été modifiés :

```
factor(species_table$species)
```

```
[1] Adelie    Gentoo    Chinstrap
Levels: Adelie Chinstrap Gentoo
```

Le premier niveau est toujours `Adélie`, puis `Chinstrap`, en enfin `Gentoo`, et non pas l'ordre du tableau nouvellement créé (`Adelie`, puis `Gentoo`, puis `Chinstrap`) car les niveaux sont toujours triés par ordre alphabétique. La conséquence est que si nous devions faire un diagramme bâtons avec ces données, la fonction `geom_col()` ne permettrait pas d'ordonner les catégories correctement :

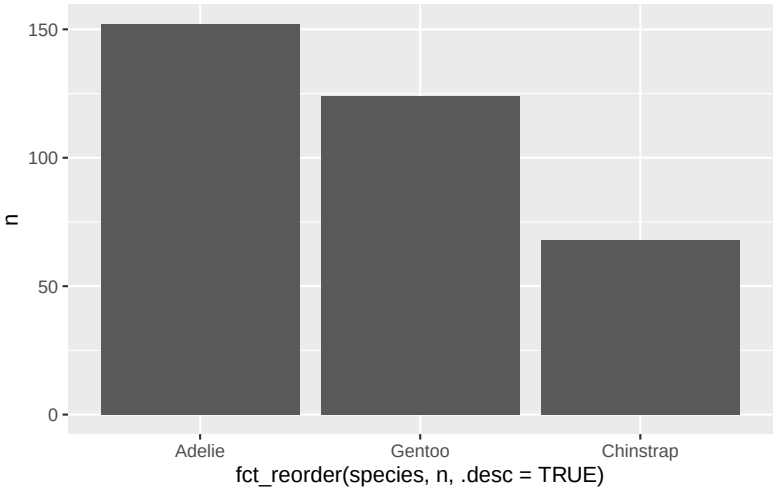
```
ggplot(species_table, aes(x = species, y = n)) +
  geom_col()
```



Si nous souhaitons trier ces catégories par effectif décroissant, la fonction `fct_infreq()` ne nous est ici d'aucune utilité. En effet, le tableau `species_table` contient une seule ligne pour chaque espèce, donc une fréquence de `1` pour chaque espèce. Le critère de la fréquence d'occurrence des modalités dans le tableau de données ne peut donc pas être utilisé. Pour parvenir à nos fins avec ce tableau déjà précompté, il faut cette fois utiliser la fonction `fct_reorder()` pour ordonner correctement les catégories. Cette fonction prends 3 arguments :

1. La variable catégorielle dont on souhaite réordonner les niveaux (ici, la variable `species` du tableau `species_table`).
2. Une variable numérique qui permet d'ordonner les catégories (ici, la variable `n` du même tableau).
3. L'argument optionnel `.desc` qui permet de préciser si le tri doit être fait en ordre croissant (c'est le cas par défaut) ou décroissant.

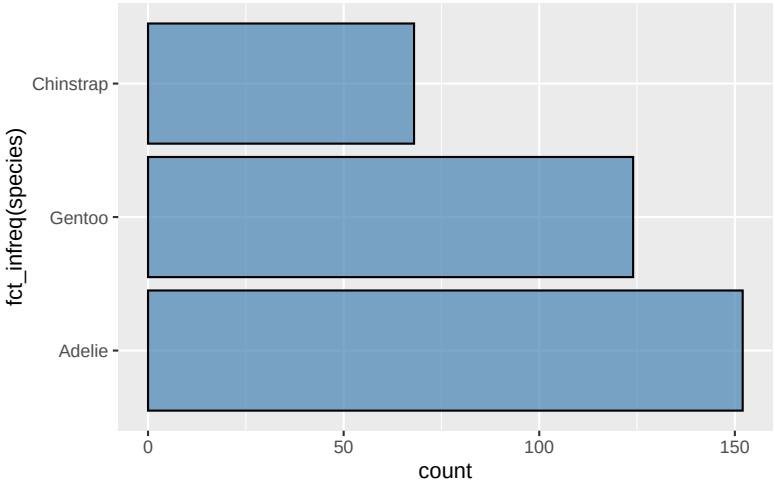
```
ggplot(species_table,
       aes(x = fct_reorder(species, n, .desc = TRUE), y = n)) +
  geom_col()
```



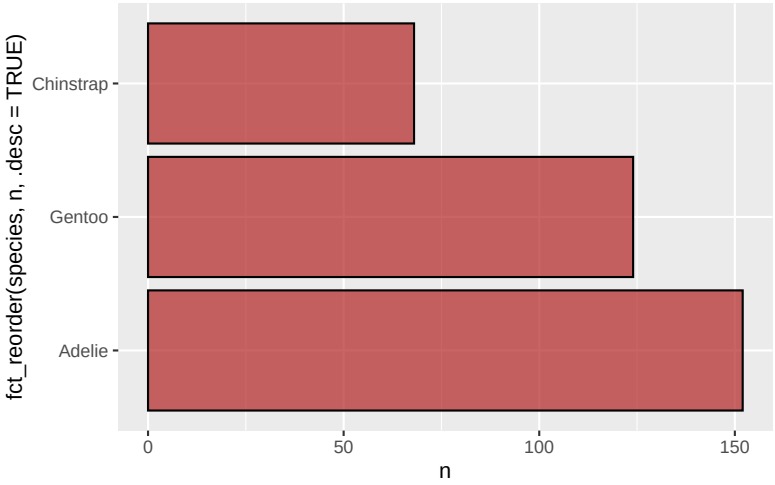
Vous voyez donc que selon le type de données dont vous disposez (soit un tableau comme `penguins`, avec toutes les observations, soit un tableau beaucoup plus compact comme `species_table`), la démarche permettant de produire un diagramme bâtons, dans lequel les catégories seront triées, sera différente.

Une dernière précision : inverser l'ordre des variables sur les axes du graphiques permet de faire un diagramme bâtons horizontal. C'est parfois très utile lorsque les modalités de la variable catégorielle sont nombreuses et/ou que leur nom est long. Faire apparaître les modalités sur l'axe des `y` au lieu de l'axe des `x` peut rendre leur lecture plus aisée :

```
ggplot(penguins, aes(y = fct_infreq(species))) +
  geom_bar(fill = "steelblue", color = "black", alpha = 0.7)
```




```
ggplot(species_table,
       aes(y = fct_reorder(species, n, .desc = TRUE), x = n)) +
  geom_col(fill = "firebrick", color = "black", alpha = 0.7)
```



3.6.1.3 Exercices

1. Quelle est la différence entre un histogramme et un diagramme bâtons ?
2. Pourquoi les histogrammes sont-ils inadaptés pour visualiser des données catégorielles ?
3. Pourquoi ne peut-on pas trier un histogramme par ordre croissant ?
4. Quelle île de l'archipel Palmer a fourni le plus grand nombre de manchots pour cette étude ?

3.6.2 Éviter à tout prix les diagrammes circulaires

À mon grand désarroi, l'un des graphiques classiquement utilisé pour représenter la distribution d'une variable catégorielle est le diagramme circulaire (ou diagramme camembert, piechart en anglais). C'est presque toujours **la plus mauvaise visualisation possible** pour représenter les effectifs ou pourcentages associés aux modalités d'une variable catégorielle. Je vous demande de l'éviter à tout prix. Notre cerveau n'est en effet pas correctement équipé pour comparer des angles et des surfaces. Ainsi, par exemple, nous avons naturellement tendance à surestimer les angles supérieurs à 90^o, et à sous-estimer les angles inférieurs à 90^o. En d'autres termes, il est difficile pour les humains de comparer des grandeurs sur des diagrammes circulaires.

À titre d'exemple, examinez ce diagramme, qui reprend les mêmes chiffres que précédemment, et tentez de répondre aux questions suivantes :

- Quelle est la catégorie la plus représentée ?
- De combien de fois la part des Gentoo mâles est-elle supérieure à celle des Chinstrap femelles ? (1,5 fois, 2 fois, 2.5 fois ?…)
- Quelle est la quatrième catégorie la plus représentée ?

Il est difficile (voir impossible) de répondre précisément à ces questions avec le diagramme circulaire de la Figure 3.13, alors qu'il est très simple d'obtenir des réponses précises avec un diagramme bâtons tel que présenté à la Figure 3.14 ci-dessous (vérifiez-le !) :

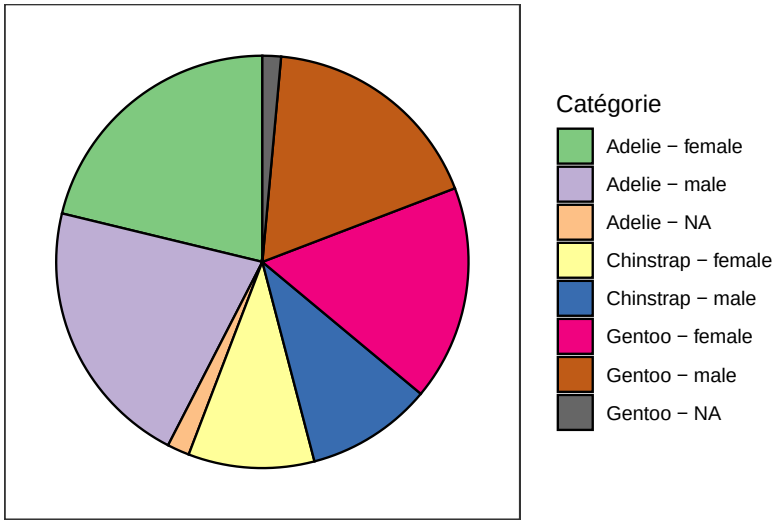


Figure 3.13: Répartition des effectifs par espèce et par sexe

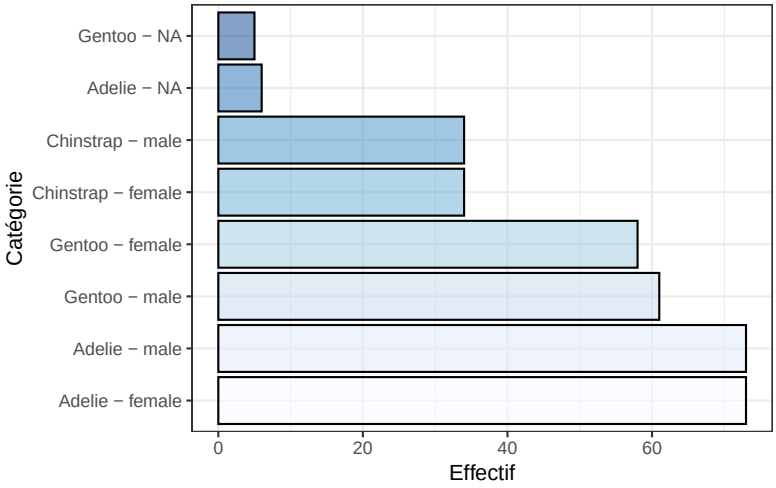


Figure 3.14: Répartition des effectifs par espèce et par sexe

3.7 Deux variables numériques

La représentation graphique la plus adaptée à la visualisation des relations entre deux variables numériques est aussi l'une des plus simples : il s'agit des nuages de points que nous avons déjà évoqués. Ici dépendant, puisque nous disposons de 2 variables numériques, nous allons en associer une à l'axe des `x` et l'autre à l'axe des `y`. Si l'on pressent que l'une des deux variables pourrait "expliquer" la seconde, ou être en partie responsable de ses variations, on l'appelle **variable explicative** et on la placera alors sur l'axe des `x`. L'autre variable, celle que l'on suppose influencée par la première est appelée **variable expliquée**, et sera associée à l'axe des `y`.

Les nuages de points de 2 variables numériques permettent donc de visualiser les relations (supposées ou réelles) entre deux variables.

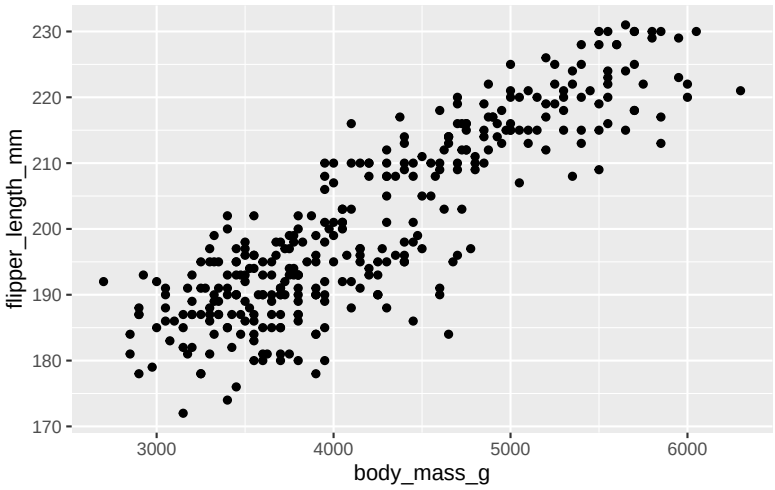
3.7.1 Nuage de points

3.7.1.1 Syntaxe élémentaire

Prenons un exemple : nous souhaitons examiner les relations qui existent entre la masse corporelle des individus et la longueur de leur nageoire. Une relation allométrique simple suppose en effet que plus un individu est grand et lourd, plus ses membres seront développés. La nature de la relation allométrique peut toutefois être radicalement différente selon les espèces. Pour l'instant, nous ne nous intéressons pas aux éventuelles différences entre espèces et nous examinerons donc l'ensemble des données, toutes espèces confondues.

```
ggplot(penguins, aes(x = body_mass_g, y = flipper_length_mm)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Ici, j'associe `body_mass_g` à l'axe des `x` car je suppose que c'est la variable explicative. Il est en effet plus logique de considérer que la masse corporelle influence la longueur des nageoires plutôt que le contraire. La variable expliquée, ici `flipper_length_mm` est associée à l'axe des `y`.

La syntaxe est donc très simple, et le graphique obtenu permet de constater que plus les individus sont lourds, plus leurs nageoires sont longues.

3.7.1.2 Droite de tendance

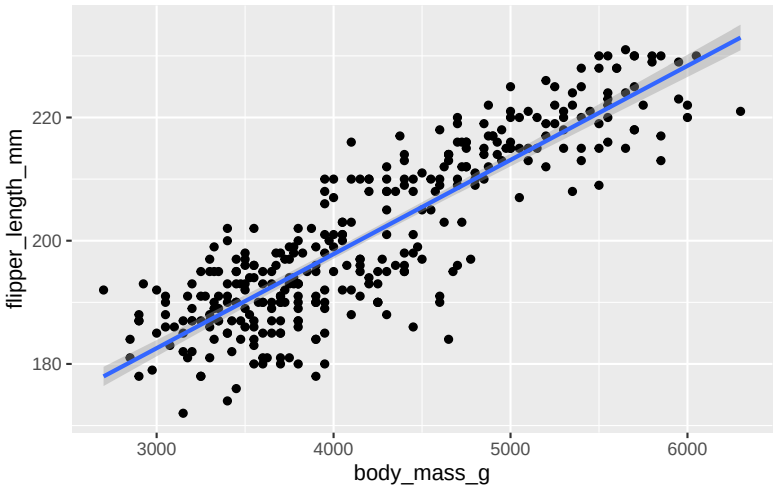
Si l'on souhaite visualiser (modéliser !) cette association entre les deux variables, on peut ajouter sur ce graphique une courbe de tendance ou une droite de régression avec l'objet géométrique `geom_smooth()` :

```
ggplot(penguins, aes(x = body_mass_g, y = flipper_length_mm)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



L'argument `method = "lm"` indique que nous souhaitons ajouter une droite de régression (`lm` est l'abréviation de "linear model"). L'intervalle grisé autour de la droite représente l'incertitude associée à la régression et indique que la "vraie" droite de régression, dans la population générale (et pas seulement dans notre échantillon de 344 individus) est probablement située dans cette zone grisée. Nous aurons l'occasion de revenir en détail sur la notion de régression linéaire et d'incertitude associée au semestre 6 de la licence SV.

3.7.1.3 Autres caractéristiques esthétiques

Comme pour tous les graphiques faisant apparaître des points, il est possible de modifier les caractéristiques esthétiques habituelles, soit en les associant à des variables du jeu de données (et en l'indiquant à l'intérieur de `aes()`), soit en les fixant à des valeurs constantes qui s'appliqueront à tous les points (et en l'indiquant alors en dehors de `aes()`). L'exemple ci-dessous illustre ces possibilités :

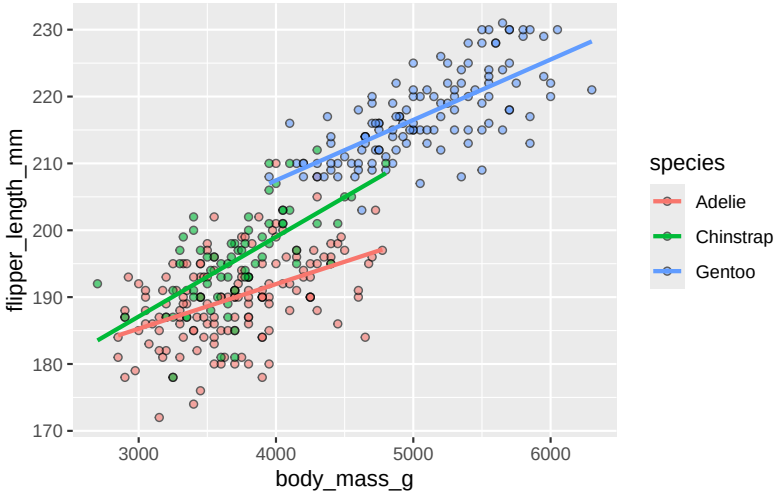
```
ggplot(penguins, aes(x = body_mass_g, y = flipper_length_mm, fill = species)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  geom_smooth(aes(color = species), method = "lm", se = FALSE)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
```

```
(`stat_smooth()`).

Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



L'argument `se = FALSE` de la fonction `geom_smooth()` permet de ne pas afficher l'intervalle d'incertitude de la régression linéaire. Ici, j'ai associé la couleur de remplissage des points et la couleur des droites de régression aux espèces (donc à l'intérieur de `aes()`, soit dans `ggplot()` soit dans `geom_smooth()`), et j'ai fixé pour tous les points, le choix du type de symbole (`shape = 21`, voir Figure 3.6), la couleur de contour (`color = "black"`) et la transparence `alpha = 0.6`.

Là encore, il ne s'agit plus strictement d'un graphique représentant les relations entre 2 variables numériques, mais bien entre 3 variables : deux variables numériques (`body_mass_g` et `flipper_length_mm`) et une variable catégorielle ou facteur (`species`). Il est finalement très simple d'ajouter d'autres variables sur un graphique bivarié tel qu'un nuage de points.

3.7.2 Les graphiques en lignes

Les graphiques en lignes, ou "linegraphs" sont généralement utilisés lorsque l'axe des `x` porte une information **temporelle**, et l'axe des `y` une autre variable numérique. Le temps est une variable naturellement ordonnée : les jours, semaines, mois,

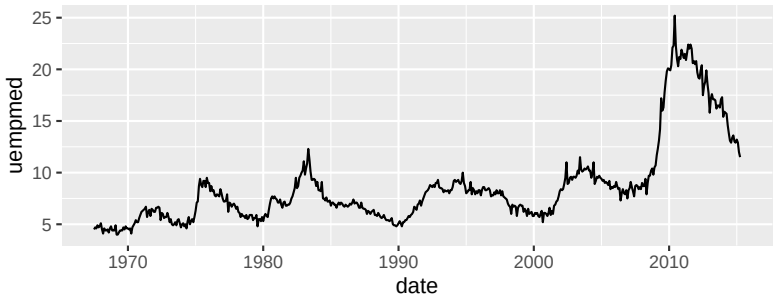
années, se suivent naturellement. Les graphiques en lignes devraient être évités lorsqu'il n'y a pas une organisation séquentielle évidente de la variable portée par l'axe des `x`. Ainsi, lorsque l'une des 2 variables dont on dispose est une variable numérique temporelle (des dates, des heures, etc.), on la place sur l'axe des `x` et la seconde variable, dont on étudiera les fluctuations au cours du temps, sur l'axe des `y`. On peut alors relier les valeurs grâce à l'objet géométrique `geom_line()` afin de créer une série temporelle. Pour illustrer cela, examinons un autre jeu de données qui contient une variable temporelle :

```
economics
```

```
# A tibble: 574 x 6
   date         pce    pop psavert uempmed unemploy
   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
 1 1967-07-01  507. 198712    12.6     4.5     2944
 2 1967-08-01  510. 198911    12.6     4.7     2945
 3 1967-09-01  516. 199113    11.9     4.6     2958
 4 1967-10-01  512. 199311    12.9     4.9     3143
 5 1967-11-01  517. 199498    12.8     4.7     3066
 6 1967-12-01  525. 199657    11.8     4.8     3018
 7 1968-01-01  531. 199808    11.7     5.1     2878
 8 1968-02-01  534. 199920    12.3     4.5     3001
 9 1968-03-01  544. 200056    11.7     4.1     2877
10 1968-04-01  544  200208    12.3     4.6     2709
# i 564 more rows
```

Le jeu de données `economics` est fourni avec le package `ggplot2`. Puisque vous avez chargé ce package (ou le `tidyverse` qui contient ce package), vous devriez pouvoir accéder à ce tableau sans difficulté. Nous nous intéresserons ici à la variable `date` que nous placerons sur l'axe des `x` et à la variable `uempmed` qui est la durée de chômage médiane dans la population américaine, en nombre de semaines, que nous placerons sur l'axe des `y`. Examinons donc comment la durée médiane du du chômage a évolué au fil du temps :

```
ggplot(economics, aes(x = date, y = uempmed)) +
  geom_line()
```



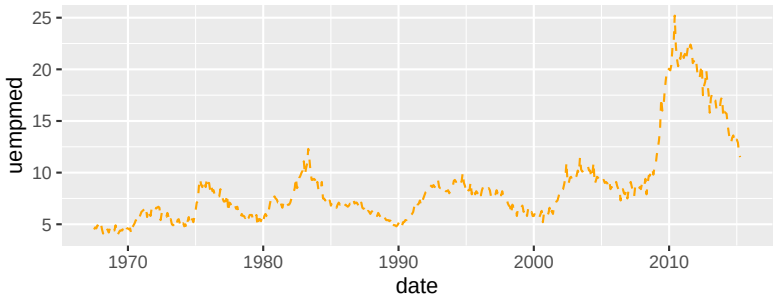
Notez que puisque la variable `date` du tableau `economics` est comprise par `R` comme étant du type "variable temporelle" (le type indiqué dans le tableau, juste sous le nom de variable, est `<date>`), l'axe des abscisses du graphique, qui est associé à cette variable, est correctement mis en forme : seules les années apparaissent.

Les graphiques en lignes permettent de visualiser des progressions/évolutions lorsqu'il existe une temporalité entre les données. Sur l'exemple, traité plus haut, du lien entre masse et longueur des nageoire des manchots, relier les points n'aurait absolument aucun sens puisque toutes les observations sont indépendantes : elles correspondent à des individus différents. Soyez donc prudents lorsque vous reliez les points dur un graphique. Cela n'est possible que lorsque les données le permettent. Vous devez donc toujours vous poser la question de la pertinence de vos choix de représentations.

Comme pour les autres types de graphiques, il est possible de modifier les caractéristiques esthétiques des lignes sur un graphique, en particulier :

- `color` : la couleur des lignes
- `size` : l'épaisseur des lignes
- `linetype` : le type de ligne (continue, pointillée, tirets, etc. Essayez plusieurs valeurs entières pour comparer les types de lignes)

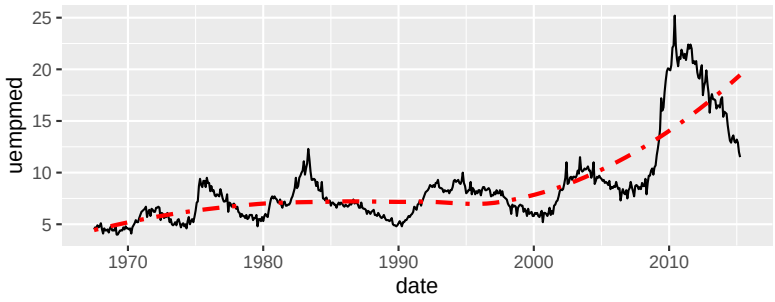
```
ggplot(economics, aes(x = date, y = uempmed)) +
  geom_line(color = "orange", linetype = 2)
```



L'argument `linetype` est également utilisable par l'objet géométrique `geom_smooth()` :

```
ggplot(economics, aes(x = date, y = uempmed)) +
  geom_line() +
  geom_smooth(se = FALSE, linetype = 4, color = "red")
```

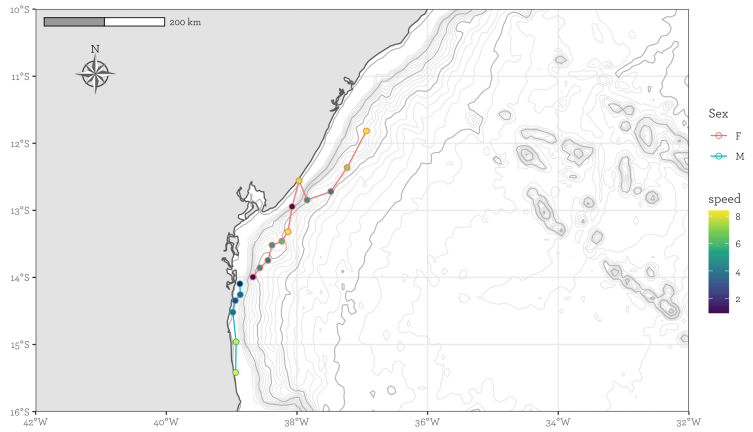
```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Globalement, la durée médiane de chômage aux USA varie de façon cyclique. La durée des cycles varie selon les période entre 5 et 10 ans environ. Depuis les années 2000, la durée de chômage a augmenté de façon très importante, pour passer de 5 à 6 semaines en 2001, à plus de 25 semaines en 2011.

3.7.3 Les cartes

Les latitudes et longitudes sont un autre type de variable numériques très particulières qui permettent notamment de produire des cartes. Il s'agit ici d'un domaine extrêmement vaste qui dépasse largement le cadre de ce livre et des cours de la licence SV. Retenez simplement qu'il est possible de produire des cartes très informatives avec `ggplot2`, et quelques autres packages spécialisés :



En règle général, les cartes portent un grand nombre de variables, numériques et/ou catégorielles. Mais tout commence toujours par 2 variables numériques, les latitudes et longitude des structures/informations que l'on souhaite représenter (traits de côte, profiles bathymétriques, lieux d'observations diverses, ...).

3.8 Deux variables catégorielles

Lorsque l'on souhaite examiner les relations entre deux variables catégorielles (ou facteurs), on a en général le choix entre les types de représentations graphiques suivants :

- les diagrammes bâtons empilés
- les diagrammes bâtons juxtaposés
- les diagrammes bâtons "facettés"
- les graphiques en mosaïque (ou mosaic plots)

Pour toutes ces méthodes, des données qui n'ont pas été comptées au préalable sont requises. Il est en effet beaucoup plus simple de travailler avec le `tidyverse` (donc avec `ggplot2`) lorsque chaque ligne d'un tableau correspond à une observation plutôt qu'à une somme d'observation. C'est le concept de **tableau rangé**, central dans le traitement de données ainsi que pour l'utilisation de tous les packages du `tidyverse`, et qui stipule qu'un tableau de données devrait contenir une unique ligne pour chaque observation, et une unique colonne pour chaque variable. Nous aurons l'occasion (notamment en L3) de voir des tableaux qui ne respectent pas ces règles et

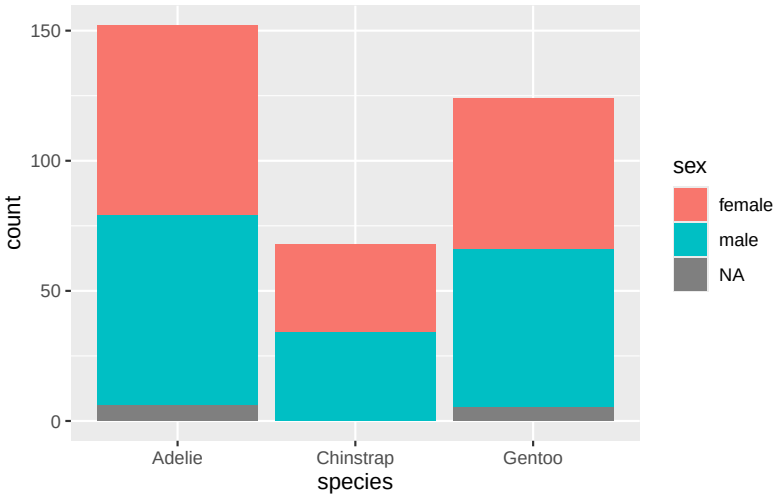
que nous devrons donc ré-organiser pour permettre leur analyse et les représentations graphiques.

Nous allons passer ces différentes possibilités en revue pour examiner les liens entre 2 variables catégorielles du jeu de données `penguins` : `species` et `sex`. La première renseigne sur l'espèce à laquelle un individu étudié appartient. La seconde renseigne sur le sexe de chaque individu. L'étude du sex-ratio est en effet souvent essentielle pour comprendre l'écologie des espèces. Les sexe-ratios sont-ils équilibrés ou non. Et s'ils ne sont pas équilibrés, sont-ils en faveur des mâles ou des femelles ?

3.8.1 Diagrammes bâtons empilés

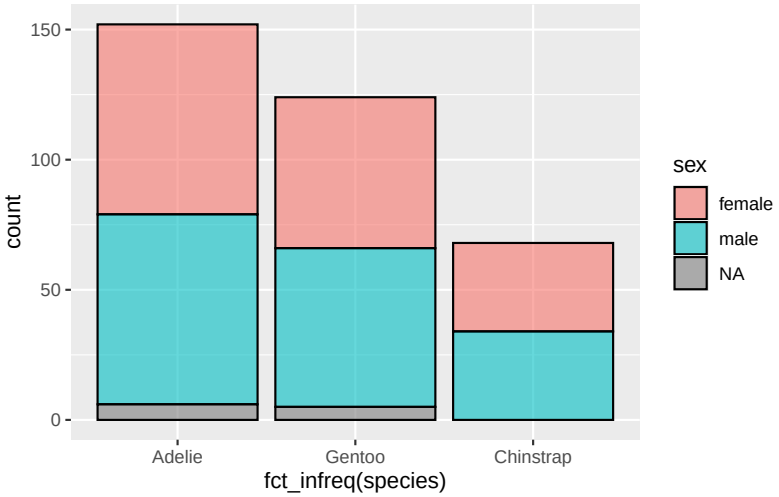
La façon la plus simple (mais rarement la meilleure) de procéder pour visualiser 2 facteurs conjointement est de créer un diagramme bâtons empilés :

```
ggplot(penguins, aes(x = species, fill = sex)) +
  geom_bar()
```



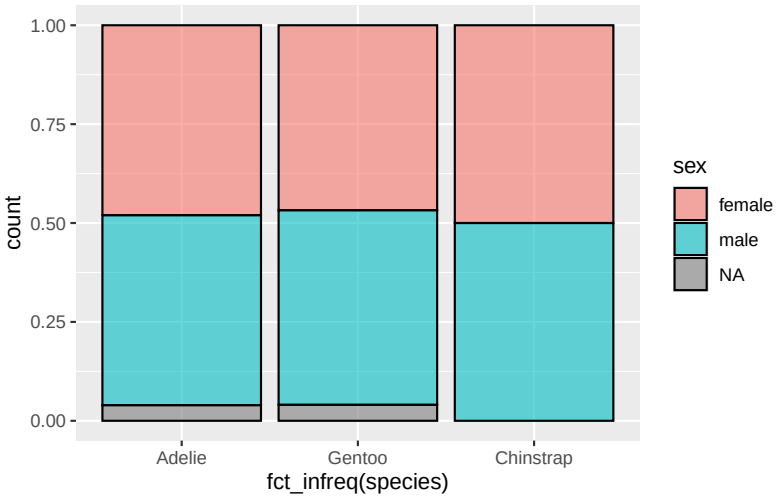
Ici, les espèces sont associées à l'axe des `x` (`x = species`) et la couleur de remplissage des barres est associée au sexe des individus (`fill = sex`), à l'intérieur de la fonction `aes()`. Comme toujours, on peut modifier certaines caractéristiques esthétiques (couleur de contour des barres, transparence, etc.) et ré-ordonner les espèces sur l'axe des abscisses :

```r
ggplot(penguins, aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black")
```



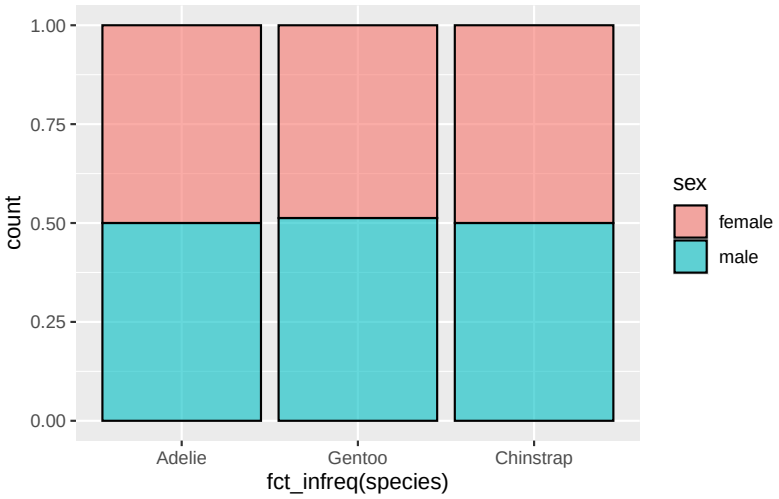
Ce type de visualisation est utile pour se rendre compte des ordres de grandeur. On voit ici clairement que l'espèce Adélie est la plus représentée dans cette étude, suivie par l'espèce Gentoo, et enfin l'espèce Chinstrap. Pour chacune de ces 3 espèces, le sex-ratio a l'air très équilibré. Toutefois, des différences subtiles de proportions entre mâles et femelles selon les espèces pourraient être masqués par les effectifs inégaux entre espèces. Il peut donc être préférable, pour comparer des proportions, de normaliser les effectifs de toutes les espèces pour ramener chaque barre du graphique à la même hauteur :

```r
ggplot(penguins, aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "fill")
```



L'argument `position = "fill"` de la fonction `geom_bar()` permet de transformer en proportions les abondances de chaque modalités de la variable portée par l'axe des `x`. L'axe des ordonnées varie maintenant entre 0 et 1 (0% et 100%), ce qui rend les comparaisons plus aisées. Ici, le fait que le sexe de quelques individus n'ait pas pu être déterminé vient gêner la lecture du graphique. On peut supprimer ces valeurs grâce à la fonction `filter()` du packages `dplyr`. Nous verrons dans le Chapitre 4 la signification du code suivant. Pour l'instant retenez simplement qu'il permet d'éliminer les individus dont le sexe est inconnu :

```
penguins |>
  filter(!is.na(sex)) |>
  ggplot(aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "fill")
```

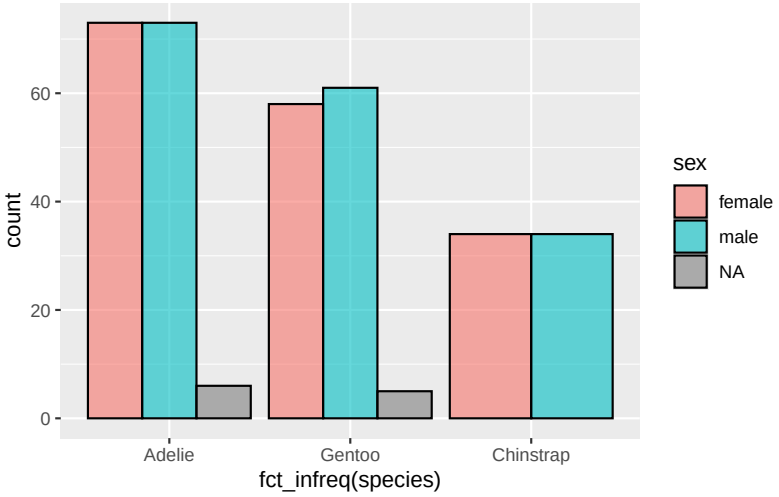


On peut maintenant constater très facilement que le sex-ratio est parfaitement équilibré pour les espèces Adélie et Chinstrap, et qu'il est très légèrement en faveur des mâles pour l'espèce Gentoo.

3.8.2 Diagrammes bâtons juxtaposés

La syntaxe permettant de produire un diagramme bâtons juxtaposé est très similaire à celle décrite ci-dessus :

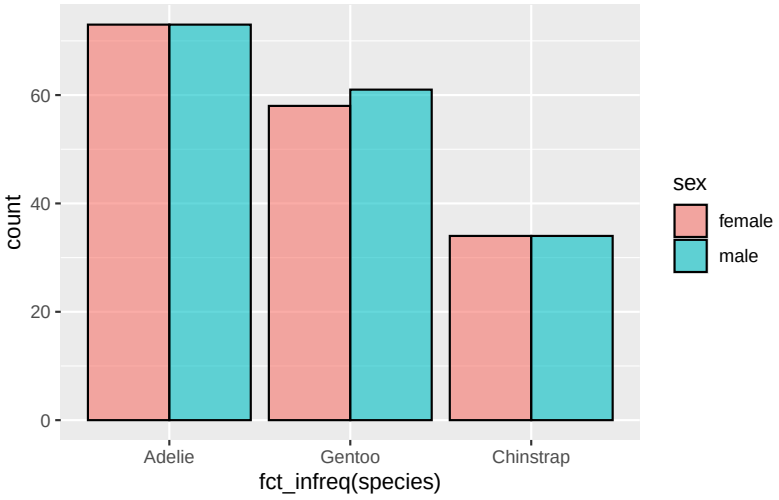
```
ggplot(penguins, aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "dodge")
```



La seule chose qui a changé est la valeur prise par l'argument `position`, que l'on fixe ici à `dodge`. L'avantage de cette représentation est qu'elle permet à la fois de visualiser les effectifs de chaque catégorie et sous-catégorie (espèce et sexe), ainsi que de comparer les proportions au sein de chaque espèce. Un inconvénient et que lorsque les catégories n'ont pas toutes le même nombre de sous-catégories, les barres ont des largeurs différentes. Ici, l'espèce Chinstrap, qui n'a que 2 sous catégories (`female` et `male`) présente des barres plus larges que les deux autres espèces qui présentent chacune 3 sous-catégories (`female`, `male` et `NA`). Pour y remédier, on peut :

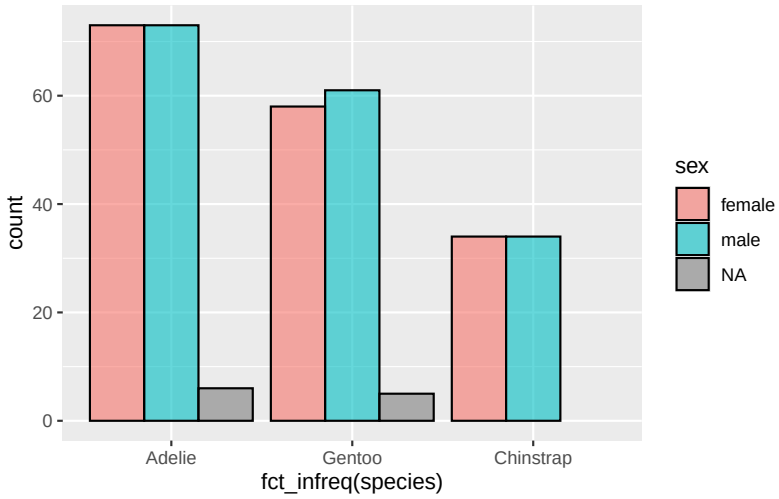
- soit retirer les données manquantes, comme précédemment :

```
penguins |>
  filter(!is.na(sex)) |>
  ggplot(aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "dodge")
```



- soit imposer que toutes les sous-catégories apparaissent pour chaque catégorie :

```
ggplot(penguins, aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black",
           position = position_dodge(preserve = "single") )
```



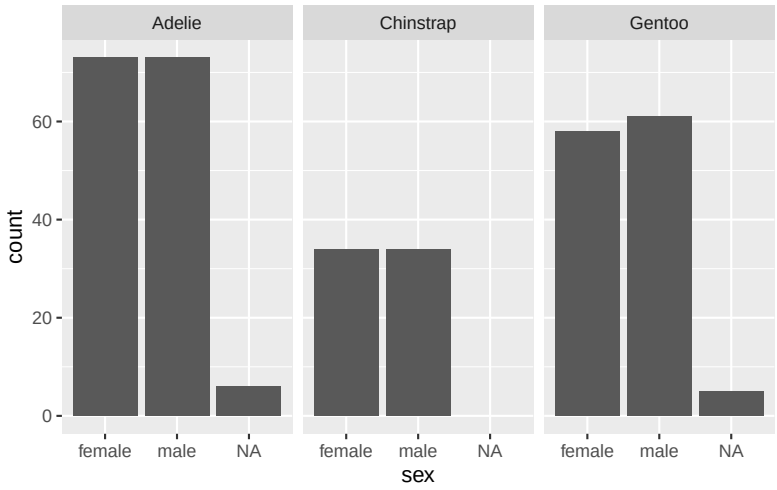
Ici, l'argument `position` prend une valeur plus complexe puisque nous faisons appel à une fonction nommée `position_dodge()`. C'est l'argument `preserve = "single"` qui permet de s'assurer que toutes les sous-catégories sont bien représentées au sein de chaque catégorie, et donc, que toutes les barres ont bien la même largeur.

Le choix d'une méthode ou de l'autre dépend de ce que l'on souhaite montrer : il n'y a pas une façon de faire meilleure ou moins bonne que l'autre. Tout dépend de l'objectif poursuivi par l'auteur du graphique.

3.8.3 Diagrammes bâtons "facettés"

Dans le jargon de `ggplot2`, les `facet`s sont simplement des sous-graphiques. Typiquement, une variable catégorielle peut être utilisée pour représenter un sous-graphique pour chaque modalité de cette variable. Ici, on peut par exemple produire un diagramme bâton pour chaque espèce, et l'axe des `x` de chaque graphique portera la variable `sex` :

```
ggplot(penguins, aes(x = sex)) +
  geom_bar() +
  facet_wrap(~species)
```

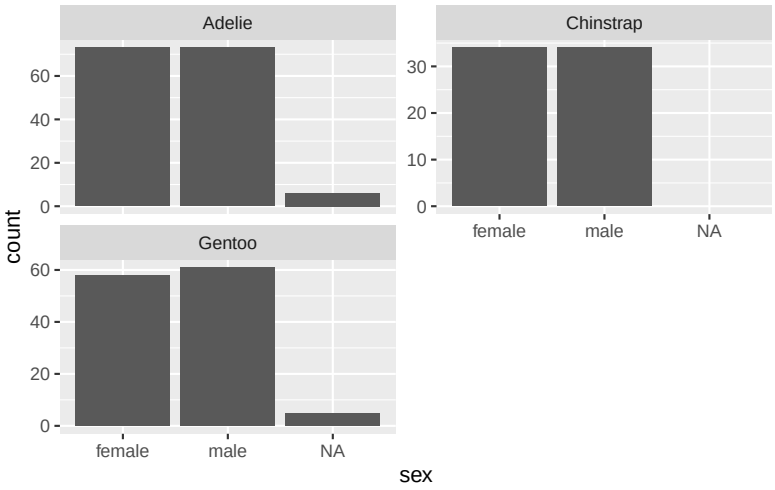


C'est la fonction `facet_wrap()` qui permet de produire plusieurs sous graphiques. Examinons quelques-une de ces particularités :

- sa syntaxe fait appel à la notion de "formule", utilisée pour certaines fonctions spécifiques dans le langage `R`. Nous en verrons des exemples en L3 pour illustrer certains tests statistiques. Le tilde `~` se lit "en fonction de". Ici `~species` signifie "crée des facets en fonction des espèces", autrement dit, produit un sous-graphique par modalité de la variable `species`.
- par défaut, les axes de tous les sous graphiques sont strictement identiques, en abscisse comme en ordonnée. On peut modifier ce comportement grâce à l'un des arguments suivants : `scales = "free_x"` (pour que les axes des abscisses soient indépendants entre les sous-graphiques), `scales = "free_y"` (pour que les axes des ordonnées soient indépendants entre les sous-graphiques) ou `scales = "free"` (pour quelles deux axes soient indépendants entre les sous-graphiques)
- l'argument `ncol =` permet de spécifier le nombre de colonnes souhaité pour l'organisation des sous-graphiques

Voici un exemple de ces syntaxes :

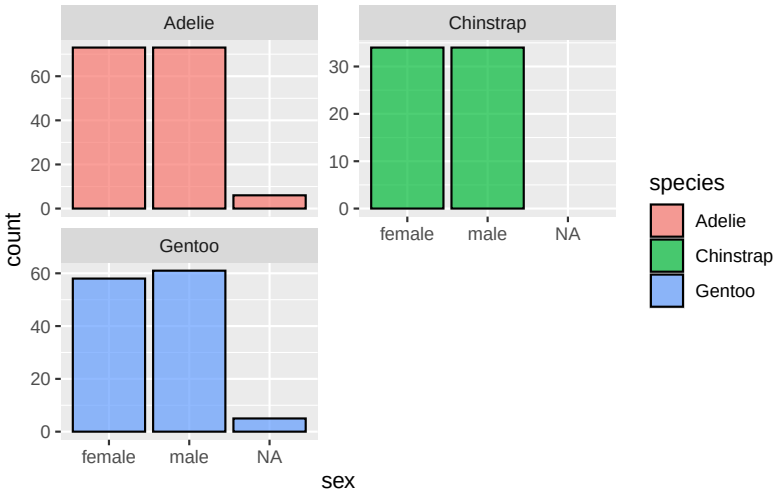
```
ggplot(penguins, aes(x = sex)) +
  geom_bar() +
  facet_wrap(~species, scales = "free_y", ncol = 2)
```



Les 3 sous-graphiques sont maintenant disposés dans 2 colonnes, et si l'axe des `x` est toujours le même pour chaque sous-graphique, les axes des `y` sont différents pour les 3 sous-graphiques.

Pour égayer un peu ce graphique, ajoutons une couleur de remplissage pour les barres, selon l'espèce :

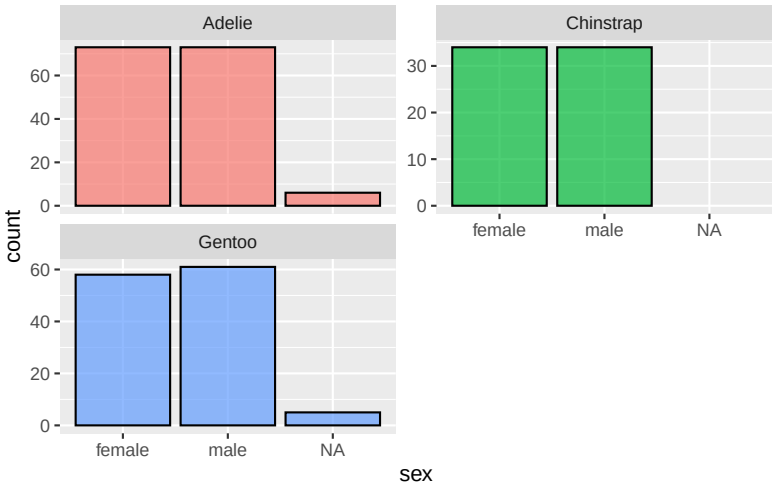
```
ggplot(penguins, aes(x = sex, fill = species)) +
  geom_bar(color = "black", alpha = 0.7) +
  facet_wrap(~species, scales = "free_y", ncol = 2)
```



La légende qui est automatiquement créée à droite est inutile puisque les sous-graphiques indiquent déjà le nom des espèces. Pour retirer une légende inutile, on peut utiliser l'argument

`show.legend = FALSE` de la plupart des objets géométriques :

```
ggplot(penguins, aes(x = sex, fill = species)) +
  geom_bar(color = "black", alpha = 0.7, show.legend = FALSE) +
  facet_wrap(~species, scales = "free_y", ncol = 2)
```



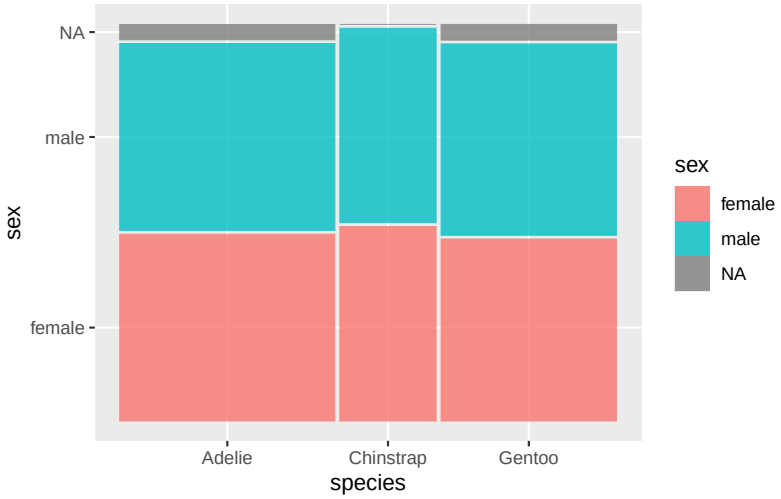
3.8.4 Mosaïc plots

Les graphiques en mosaïque sont une alternative aux diagrammes bâtons en tous genre. Ils permettent de visualiser à la fois les effectifs et de comparer les proportions. La difficulté de ce genre de graphique est qu'il n'existe pas d'objet géométrique permettant de les représenter simplement dans le package `ggplot2`. Le package `ggmosaic` de Jeppson, Hofmann, et Cook (2021) est toutefois entièrement dédié à ce type de graphique. Installez ce package puis chargez-le en mémoire :

```
install.packages("ggmosaic")
library(ggmosaic)
```

On peut maintenant accéder à un nouvel objet géométrique, `geom_mosaic()`, dont l'utilisation est un peu différente de celle que nous avons vu jusqu'ici :

```r
ggplot(penguins) +
  geom_mosaic(aes(x = product(species), fill = sex))
```



Il faut obligatoirement :

1. spécifier `aes()` à l'intérieur de `geom_mosaic()` et non à l'intérieur de `ggplot()`
2. utiliser la fonction `product()` (qui fait elle aussi partie du package `ggmosaic`) pour indiquer quelle variable catégorielle on souhaite associer à l'axe des `x`
3. Comme pour les diagrammes bâtons, la couleur de remplissage est associée à la seconde variable catégorielle de façon tout à fait classique

Comme pour les diagrammes en bâtons empilés pour lesquels on spécifie `position = "fill"`, toutes les barres d'un graphique en mosaïque ont la même hauteur, ce qui permet de visualiser les proportions de chaque sexe pour chaque espèce, mais pas les effectifs. C'est ici la largeur des barres qui est proportionnelle aux effectifs de chaque espèce. Si on n'accède par directement aux valeurs absolues, on peut néanmoins effectuer des comparaisons d'ordres de grandeur. L'espèce Adélie est ainsi la plus représentée dans nos données, suivie de l'espèce Gentoo puis de l'espèce Chinstrap.

Au final, le choix d'un graphique doit vous permettre de mettre en évidence les relations qui vous paraissent importantes de la façon la plus visuelle et évidente possible pour une personne ne connaissant pas vos données. Votre choix

dépendra donc des données disponibles et de votre objectif (p. ex. comparaisons de proportions ou de valeurs absolues, nombreuses modalités ou seulement quelques unes, etc.).

3.9 Une variable de chaque type

Les représentations graphiques réalisables et pertinentes lorsque l'on dispose d'une variable numérique et d'un facteur sont souvent des adaptations des graphiques précédents. Globalement, trois choix s'offrent à nous :

1. les histogrammes facettés
2. les stripcharts
3. les boîtes à moustaches, que nous détaillerons au semestre 4. Nous donnerons ici un simple exemple sans expliquer la signification de tous les éléments de ces graphiques

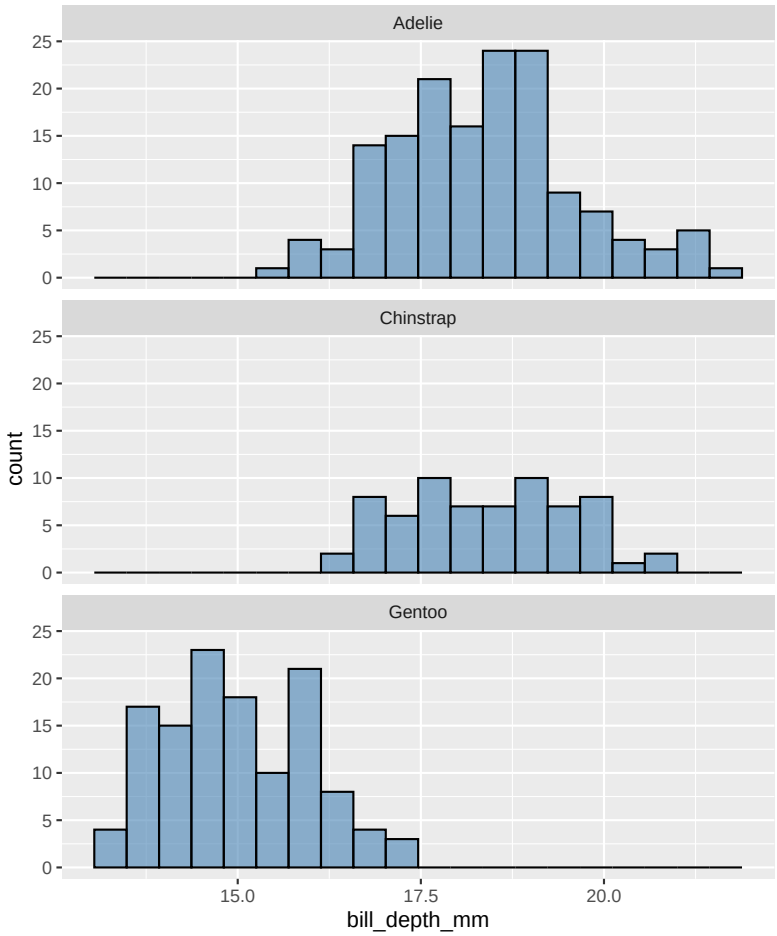
Pour illustrer ces différentes possibilités, intéressons nous maintenant à la relation qui existe entre l'épaisseur du bec des manchots (`bill_depth_mm`, variable numérique) et l'espèce (`species`, variable catogorielle ou facteur)

3.9.1 Histogrammes "facettés"

La syntaxe est ici tout à fait classique. Pour réaliser un histogramme, on place la variable numérique sur l'axe des abscisses. La variable catégorielle nous servira à créer les sous graphiques, ici, un par espèce. Afin de faciliter les comparaisons, nous placerons les sous-graphiques les uns sous les autres en spécifiant `ncol = 1`. Enfin, l'aspect général sera amélioré en modifiant quelques caractéristiques esthétiques :

```
ggplot(penguins, aes(x = bill_depth_mm)) +
  geom_histogram(fill = "steelblue", color = "black",
                 alpha = 0.6, bins = 20) +
  facet_wrap(~species, ncol = 1)
```

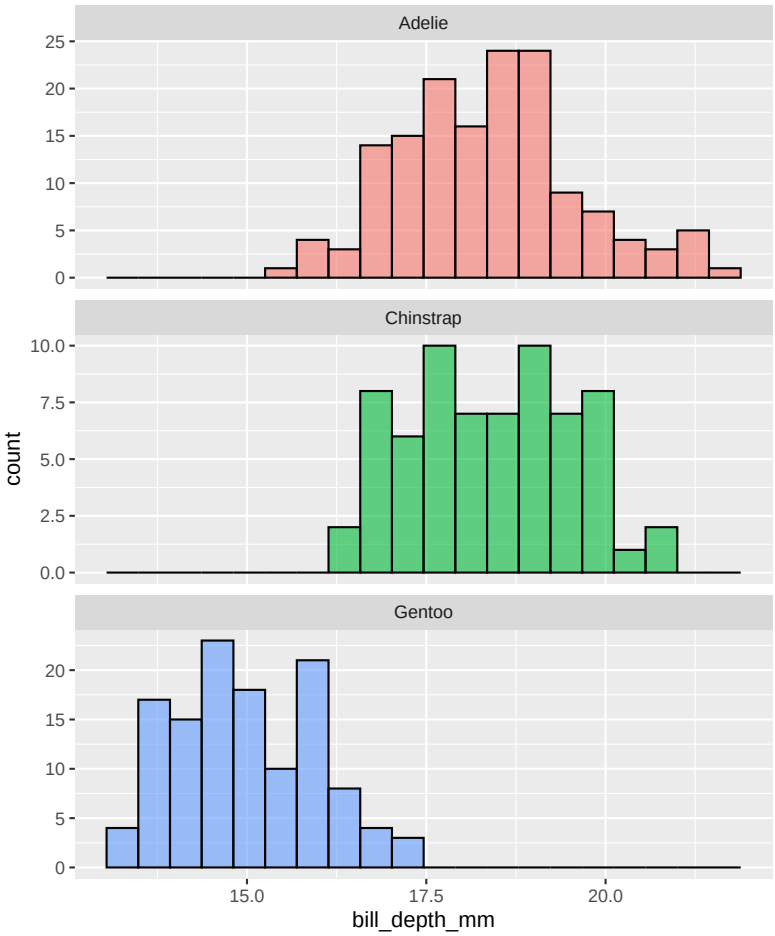
```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



On peut aussi choisir d'utiliser une couleur pour chaque espèce (mais on n'affichera pas la légende puisque les espèces sont déjà séparées dans les sous graphiques). En outre, puisque les effectifs des Chinstrap sont bien plus faibles que pour les deux autres espèces, on a intérêt à "libérer" l'axe des y afin que l'histogramme des Chinstrap soit plus facilement lisible (il apparaît pour l'instant très "écrasé" comparé aux autres).

```
ggplot(penguins, aes(x = bill_depth_mm, fill = species)) +
  geom_histogram(show.legend = FALSE, color = "black",
                 alpha = 0.6, bins = 20) +
  facet_wrap(~species, ncol = 1, scales = "free_y")
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_bin()`).
```



Les Gentoo, qui ont pourtant des masses corporelles supérieures à celle des deux autres espèces (voir Figure 3.5 de la Section 3.5.2), ont visiblement des becs moins épais (entre 12 et 17 mm) que les deux autres espèces (entre 16 et 22 mm).

> **!** Important
>
> C'est la position des données le long de l'axe des `x` qui nous permet de faire des comparaisons pertinentes. Il est donc essentiel de présenter les différents histogrammes les uns sous les autres, en conservant la même échelle pour les abscisses de tous les sous-graphiques.

Ici, on peut donc discuter de la distribution de la variable numérique pour chaque modalité de la variable catégorielle (*i.e.* quelle distribution de l'épaisseur des becs pour chaque espèce), mais on peut en plus faire des comparaisons entre

modalités (entre les espèces). Cela est beaucoup plus pertinent que de s'intéresser à la distribution de l'épaisseur des becs toutes espèces confondues.

3.9.2 Les stripcharts

Nous avons déjà abordé ce type de graphique dans la Section 3.5.2. Contrairement à la situation où nous n'avions qu'une variable numérique et où nous devions fixer `x = ""` pour que toutes les observations se placent au même niveau de l'axe des abscisses (voir Figure 3.8), nous allons ici associer la variable catégorielle à l'axe des `x`. La variable numérique sera quant-à-elle toujours associée à l'axe des ordonnées :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

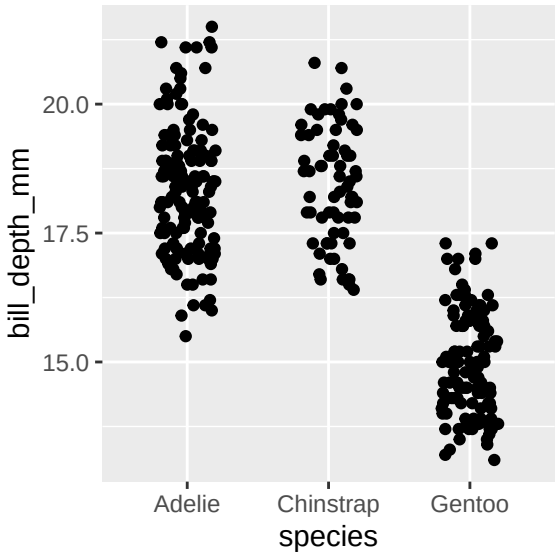


Figure 3.15: Un exemple de stripchart

Notez que la position des points sur l'axe des `y` doit parfaitement correspondre aux valeurs contenues dans le jeu de

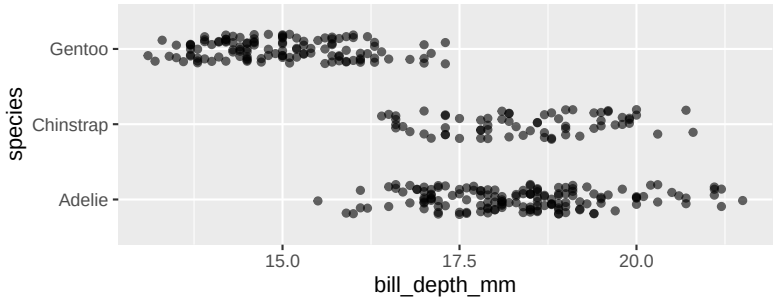
données pour la variable numérique d'intérêt. Cela signifie que l'argument `height` doit obligatoirement être fixé à 0.

Comme pour les diagrammes bâtons, il est possible de produire des stripcharts horizontaux. Les modifications à apporter sont alors les suivantes :

- la variable numérique est associée à l'axe des `x`
- la variable catégorielle est associée à l'axe des `y`
- la dispersion horizontale `width` doit obligatoirement être fixée à 0
- la dispersion verticale `height` doit être comprise entre 0.1 et 0.4 pour étaler les points de chaque modalité et ainsi éviter l'overplotting

```
ggplot(penguins, aes(x = bill_depth_mm, y = species)) +
  geom_jitter(width = 0, height = 0.20, alpha = 0.6)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

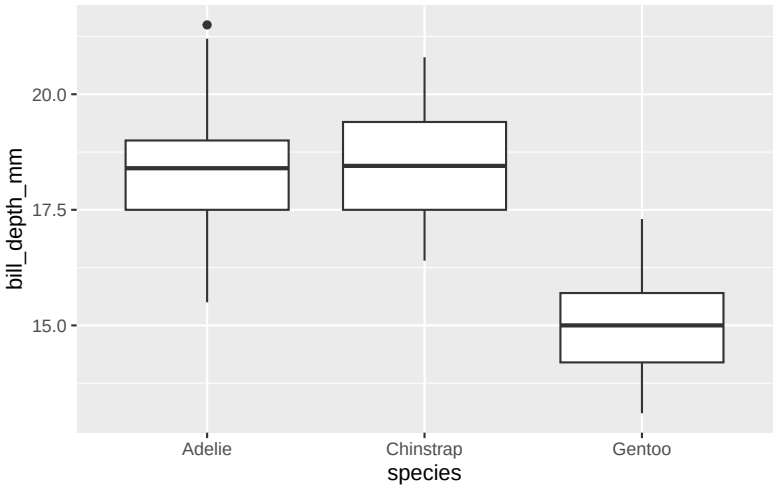


3.9.3 Les boîtes à moustaches ou boxplots

Voilà à quoi ressemble un graphique de ce type pour les données qui nous intéressent (épaisseur des becs selon l'espèce) :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_boxplot()
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_boxplot()`).
```



Dans la forme, ça ressemble un à un stripchart (comparez par exemple avec la syntaxe et les résultats obtenus à la Figure 3.15). Néanmoins, ici, au lieu de visualiser tous les points du jeu de données, seules quelques valeurs caractéristiques sont utilisées pour construire le boîte à moustache de chaque espèce. Les différents éléments d'un boxplot, sont les suivants :

- La limite inférieure de la boîte correspond au premier quartile : 25% des données de l'échantillon sont situées au-dessous de cette valeur.
- La limite supérieure de la boîte correspond au troisième quartile : 25% des données de l'échantillon sont situées au-dessus de cette valeur.
- Le segment épais à l'intérieur de la boîte correspond au second quartile : c'est la médiane de l'échantillon. 50% des données de l'échantillon sont situées au-dessus de cette valeur, et 50% au-dessous.
- La hauteur de la boîte correspond à ce que l'on appelle l'étendue inter-quartile ou Inter Quartile Range (IQR) en anglais. On trouve dans cette boîte 50% des observations de l'échantillon. C'est une mesure de la dispersion des 50% des données les plus centrales. Une boîte plus allongée indique donc une plus grande dispersion.
- Les moustaches correspondent à des valeurs qui sont en dessous du premier quartile (pour la moustache du bas) et au-dessus du troisième quartile (pour la moustache du haut). La règle utilisée dans `R` est que ces moustaches s'étendent jusqu'aux valeurs minimales et maximales de l'échantillon, mais elles ne peuvent en aucun

cas s'étendre au-delà de 1,5 fois la hauteur de la boîte (1,5 fois l'IQR) vers le haut et le bas. Si des points apparaissent au-delà des moustaches (vers le haut ou le bas), ces points sont appelés "outliers". On peut en observer un pour l'espèce Adélie. Ce sont des points qui s'éloignent du centre de la distribution de façon importante puisqu'ils sont au-delà de 1,5 fois l'IQR de part et d'autre du premier ou du troisième quartile. Il peut s'agir d'anomalies de mesures, d'anomalies de saisie des données, ou tout simplement, d'enregistrements tout à fait valides mais atypiques ou extrêmes. J'attire votre attention sur le fait que la définition de ces outliers est relativement arbitraire. Nous pourrions faire le choix d'étendre les moustaches jusqu'à 1,8 fois l'IQR (ou 2, ou 2,5). Nous observerions alors beaucoup moins d'outliers. D'une façons générale, la longueur des moustaches renseigne sur la variabilité des données en dehors de la zone centrale. Plus elles sont longues, plus la variabilité est importante. Et dans tous les cas, l'examen attentif des outliers est utile car il nous permet d'en apprendre plus sur le comportement extrême de certaines observations.

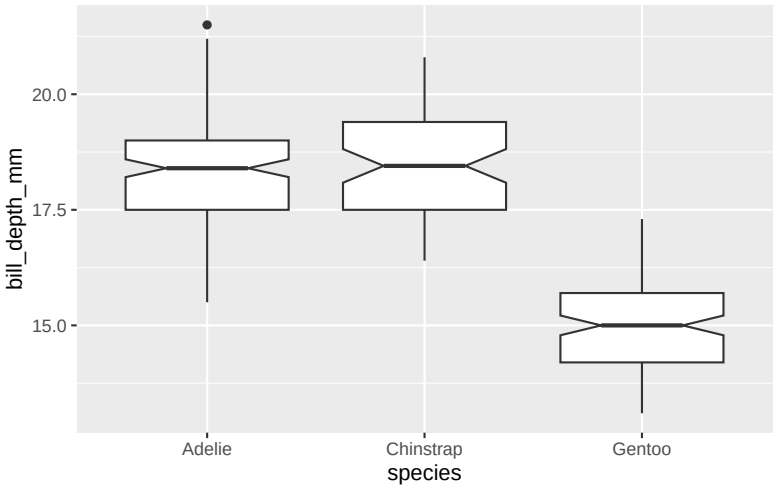
Lorsque les boîtes ont une forme à peu près symétrique de part et d'autre de la médiane (c'est le cas pour notre exemple), cela signifie qu'un histogramme des mêmes données serait symétrique également (on peut le vérifier avec les histogrammes de la Section 3.9.1).

3.9.3.1 L'intervalle de confiance à 95% de la médiane

On peut également ajouter une encoche autour de la valeur de médiane en ajoutant l'argument `notch = TRUE` à la fonction `geom_boxplot()` :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_boxplot(notch = TRUE)
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_boxplot()`).
```



L'encoche qui apparaît sur chaque boîte à moustache correspond à l'étendue de l'intervalle de confiance à 95% de la médiane. Pour chaque échantillon, nous espérons que la médiane calculée soit le reflet fidèle de la vraie valeur de médiane de la population générale. Mais il sera toujours impossible d'en avoir la certitude absolue. Le mieux que l'on puisse faire, c'est quantifier l'incertitude associée à l'estimation de la médiane à partir des données d'un échantillon. L'intervalle de confiance nous indique qu'il y a de bonnes chances que la vraie valeur de médiane de la population générale (qui restera à jamais inconnue) se trouve dans cet intervalle.

Nous reviendrons sur cette notion importante plus tard dans le cursus, car ce type de graphique nous permettra d'anticiper sur les résultats des tests statistiques de comparaison de moyennes.

Au final, nous avons 3 moyens d'obtenir des informations de distribution :

- observer l'ensemble des données brutes grâce à un nuage de points ou stripchart
- regrouper en partie les données brutes dans les classes d'un histogramme. On ne visualise plus l'ensemble des données individuelles, mais un résumé de ces données puisqu'on ne dispose plus que d'une unique valeur pour chaque classe de l'histogramme. L'histogramme peut donc résumer des centaines voire des milliers de points sous la forme d'un petit nombre de classes (entre 10 et 40 en général)

- regrouper très fortement les données brutes sous la forme d'une boîte à moustache. Les boîtes à moustaches permettent de résumer l'information de centaines ou milliers de points sous la forme d'un résumé statistique composé de 5 valeurs (minimum et maximum, médiane, premier et troisième quartiles), ou 7 si l'on ajoute les encoches des intervalles de confiance à 95% des médianes. On observe alors moins facilement les nuances subtiles de distribution qu'avec un histogramme ou les données brutes, mais l'avantage est qu'on peut comparer facilement les grandes tendances d'un grand nombre de séries de données (parfois plusieurs dizaines) en plaçant des boîtes à moustaches côte à côte.

La Figure 3.16 illustre ces 3 possibilités de visualisation de la distribution d'une variable numérique (ici, la distribution des masses corporelles des manchots Adélie) :

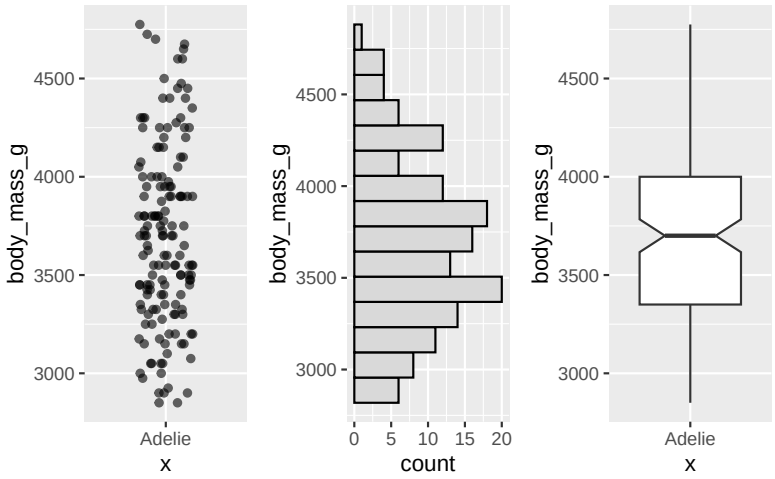


Figure 3.16: Trois façons de visualiser la distribution des masses des manchots Adélie

3.10 Trois variables (et plus !)

Lorsque l'on dispose de 3 variables, les situations possibles commencent à être nombreuses :

- trois variables numériques
- deux variables numériques et un facteur
- une variable numérique et deux facteurs

- trois facteurs

Pour chacune de ces situations, on peut en générale reprendre les types de graphiques proposés dans les 3 sections précédentes consacrées aux situations où l'on dispose de 2 variables (), et :

- soit ajouter une variable sous forme de code couleur (avec `color` ou `fill` à l'intérieur de `aes()`)
- soit ajouter une variable sous forme de `facets` (avec `facet_wrap()` ou avec `facet_grid()`)

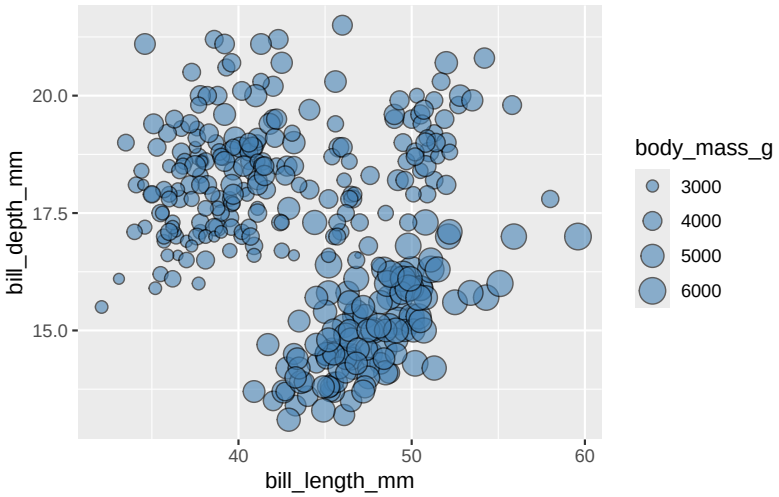
Les possibilités sont très nombreuses et il ne sera pas possible d'être exhaustif ici. Je fournis néanmoins quelques exemples ci-dessous afin que vous compreniez bien la logique. Ensuite, ça sera à vous d'expérimenter selon les données dont vous disposez, les questions scientifiques que vous vous posez, et les relations que vous souhaitez explorer/visualiser.

3.10.1 Trois variables numériques

Dans cette situation, on fait en général un nuage de points qui porte une variable numérique sur chaque axe, et on associe la troisième variable numérique soit à la couleur des points, soit à leur taille (soit aux deux à la fois). Par exemple, pour examiner les relations entre longueur du bec, épaisseur du bec, et masse corporelle, on peut procéder ainsi :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     size = body_mass_g)) +
  geom_point(shape = 21, fill = "steelblue", color = "black", alpha = 0.6)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

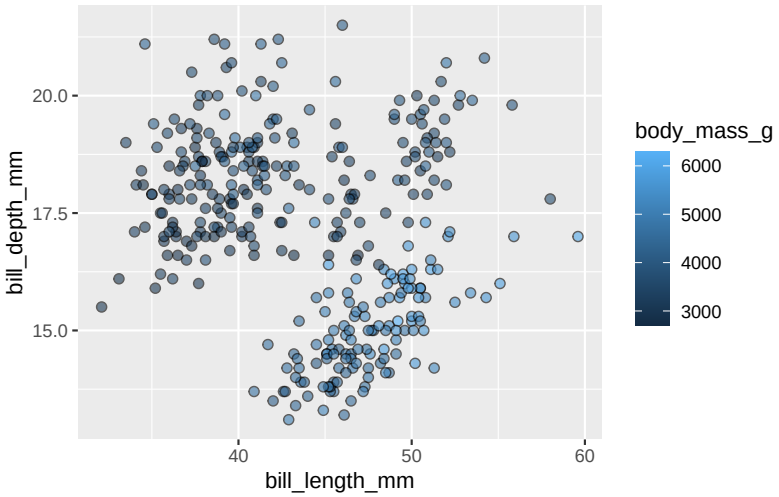


C'est ce qu'on appelle un "bubble plot". Ici on constate que les individus qui ont les becs les plus courts, sont aussi ceux qui ont un bec épais (groupe de points en haut à gauche). Ces individus sont parmi les plus légers (symboles de petite taille). À l'inverse, les individus ayant les becs les plus longs ont aussi des becs peu épais (groupes de points situés en bas à droite). Ces individus sont parmi les plus lourds du jeu de données (symboles de grandes taille).

Une autre façon de visualiser ces mêmes données consiste à associer la masse des individus à la couleur de remplissage des symboles :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6, size = 2)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

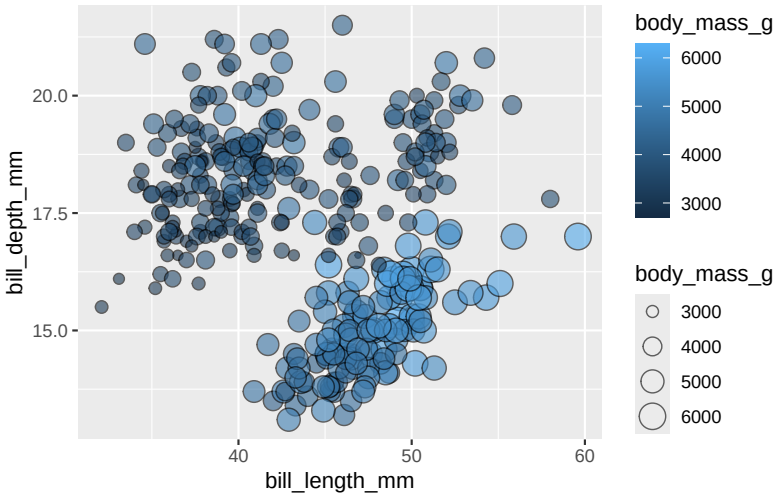


Cette fois, les individus les plus légers apparaissent en bleu très sombre, et les individus les plus lourds en bleu très clair. Ce choix de couleur nous est imposé, mais nous verrons plus loin comment le modifier pour rendre ce type de graphique plus facile à lire. Lorsque nous associons une variable numérique continue à la couleur des points, la légende qui est générée automatiquement pour nous par R sera toujours un gradient de couleurs. Si vous revenez en arrière au niveau des graphiques en mosaïques (Section 3.8.4), ou au niveau des diagrammes bâtons juxtaposés (Section 3.8.2), vous verrez que lorsque la couleur est associée à une variable catégorielle (ou facteur), la légende présente des couleurs distinctes, une pour chaque modalité du facteur considéré. Là encore, R choisit les couleurs pour nous. Mais là encore, nous verrons comment imposer des couleurs différentes si les choix par défaut ne nous conviennent pas.

Enfin, il est évidemment possible de jouer à la fois sur la couleur et sur la taille des symboles :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Ici, l'information de masse est donc associée à 2 caractéristiques esthétiques distinctes : la couleur de remplissage des points et leur taille. Cela rend la lecture plus facile dans certaines situations.

Au final, nous avons donc associé 3 variables numériques à 4 caractéristiques esthétiques du graphique :

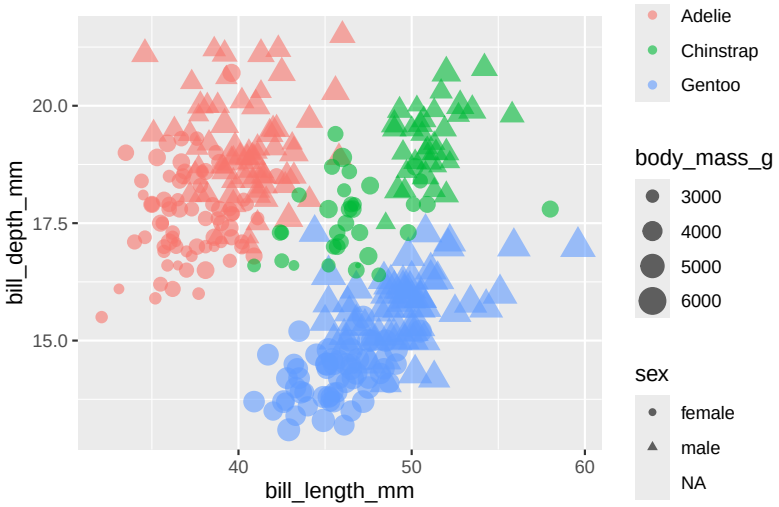
- `bill_length_mm` est associée à `x`
- `bill_depth_mm` est associée à `y`
- `body_mass_g` est associé à `fill`
- `body_mass_g` est associé à `size`

Rien ne nous empêche d'ajouter des variables et des caractéristiques esthétiques. C'est ce que nous allons voir tout de suite.

3.10.2 Cinq variables !

Pour commencer, essayez de reproduire le graphique suivant :

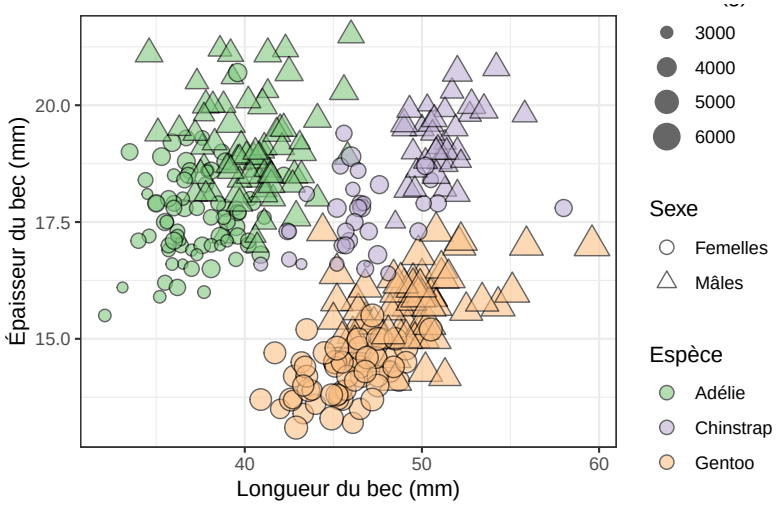
```
Warning: Removed 11 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Ici, 5 variables du jeu de données (3 numériques et 2 facteurs) sont associées à 5 caractéristiques esthétiques du graphique. Le graphique est donc très riche, on peut voir par exemple :

- que les 3 espèces ont des morphologies de bec assez distinctes : les Gentoo ont des becs longs et fins, les Chinstrap ont des becs longs et épais, et les Adélie ont des becs courts et épais.
- qu'un dimorphisme sexuel est présent au niveau du bec : pour chaque espèce, les mâles ont des becs plus longs et épais que les femelles
- qu'un dimorphisme sexuel est présent au niveau des masses : pour chaque espèce, les mâles sont plus lourds que les femelles

Au final, beaucoup d'informations sont présentées sur ce graphique et c'est presque trop. Même en améliorant l'aspect général du graphique pour le rendre plus lisible (voir ci-dessous), il vaut parfois mieux se limiter à 2 ou 3 variables et faire plusieurs graphiques, plutôt que de tout mettre sur le même. Une bonne solution consiste souvent à mettre 2 ou 3 variables sur un graphique, mais à faire plusieurs sous-graphiques pour chaque modalité d'une 4ème et/ou d'une 5ème variable catégorielle.



En particulier, sur ce graphique, il est presque impossible de déterminer la masse des individus mâles. La légende indique en effet des tailles de cercles qui correspondent à des masses spécifiques. Mais nous n'avons aucune indication pour la taille des triangles. Il vaudrait donc mieux procéder ainsi :

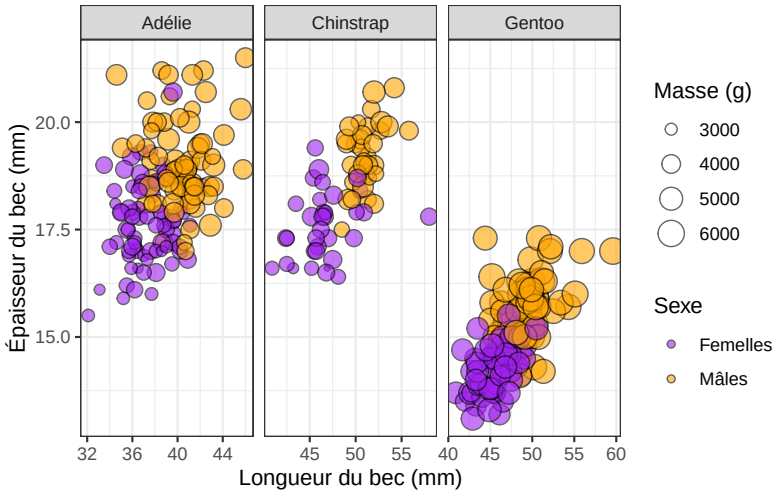


Figure 3.17: Relation entre la morphologie du bec, la masse et le sexe chez trois espèces de manchots de l'archipel Palmer

En associant le sexe des individus à la couleur de remplissage plutôt qu'à la forme des points, et en faisant un sous-graphique par espèce, on élimine la difficulté de lecture liée à la taille des symboles triangulaires. L'information concernant la masse des individus est donc plus facile à visuali-

ser. Le dimorphisme sexuel de taille des becs, présent pour chaque espèce, apparaît beaucoup plus clairement qu'avant (les mâles ont des becs plus longs et épais que les femelles). Mais les **différences inter-spécifiques** de morphologie des becs sont moins visibles qu'avant, notamment pour les différences de longueur des becs selon les espèces. On ne peut malheureusement pas gagner sur les tableaux à la fois. C'est la raison pour laquelle les choix de graphiques que vous ferez devront refléter les questions auxquelles vous vous intéressez, et les messages que vous souhaitez faire passer.

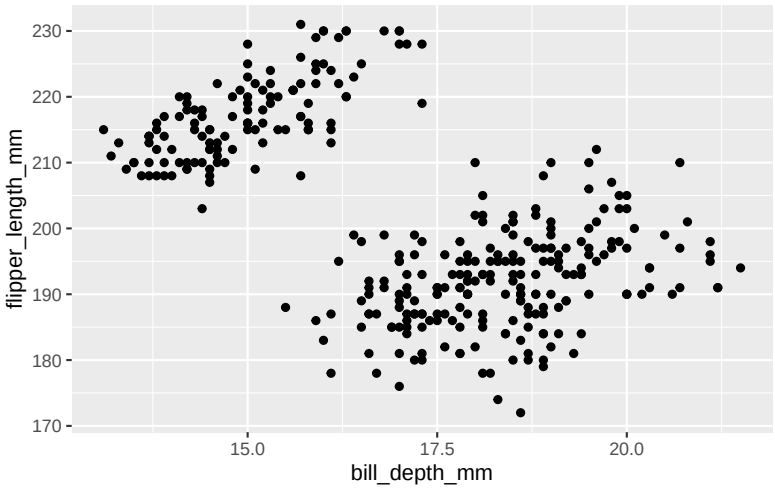
3.10.3 Deux variables numériques et un facteur

L'exemple qui suit est fondamental pour bien comprendre l'importance d'explorer en détail tous les aspects d'un jeu de données pour éviter de dire de grosses bêtises.

Imaginez que dans le jeu de données `penguins`, on souhaite étudier la relation qui existe entre l'épaisseur du bec des individus et la longueur des nageoires. Ces deux variables étant numériques, il semble logique de commencer par faire un nuage de points :

```
ggplot(penguins, aes(x = bill_depth_mm, y = flipper_length_mm)) +
  geom_point()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



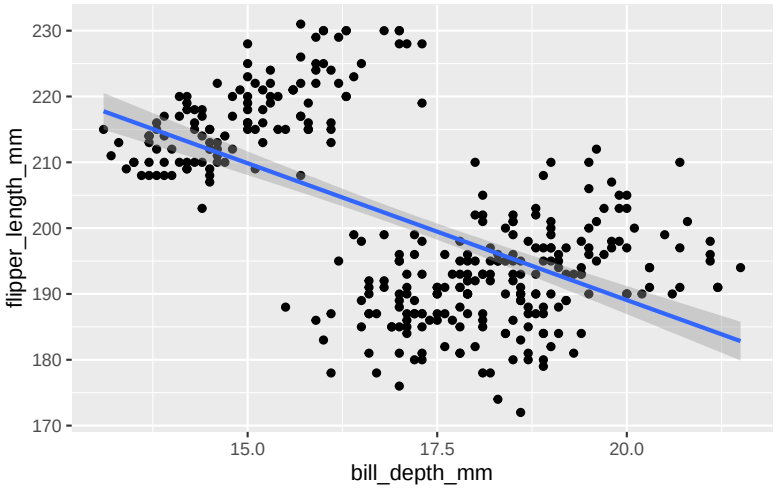
Nous avons vu plus haut que pour visualiser le relations qui existent entre deux variables numériques, il est possible d'ajouter une courbe ou une droite de tendance avec la fonction `geom_smooth()` :

```
ggplot(penguins, aes(x = bill_depth_mm, y = flipper_length_mm)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Si l'on s'en tient à ça, la relation semble claire : plus le bec des individus est épais, plus leurs nageoires sont courtes, et inversement. En outre, nous avons visiblement deux groupes d'individus qui présentent des caractéristiques distinctes : certains ont des becs fins et des nageoires très longues, quand d'autres ont des becs épais et des nageoires courtes. Et quasiment aucun individu ne présente de caractéristiques intermédiaires (becs d'épaisseur moyenne épais et nageoires moyennes).

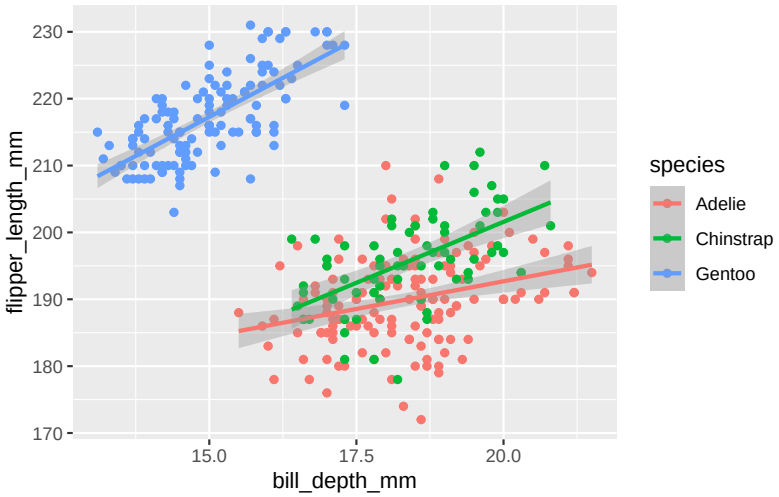
En réalité, cette vision des choses est totalement trompeuse ! N'oubliez pas que nous avons 3 espèces distinctes dans ce jeu de données, et que ces espèces peuvent présenter des caractéristiques morphologiques très variées. Examiner la relation becs-nageoires tel que nous l'avons fait, sans considérer les espèces, n'a strictement aucun sens ! Pour s'en convaincre, il suffit d'associer la couleur (des points et des lignes) à l'espèce :

```
ggplot(penguins, aes(x = bill_depth_mm, y = flipper_length_mm,
                     color = species)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Le résultat obtenu ici est à l'opposé de nos conclusions précédentes : la relation entre les deux variables numérique est en fait positive ! Au sein de chaque espèce, les individus possédant les becs les plus épais sont aussi ceux qui possèdent les nageoires les plus longues !

En statistiques ce phénomène (observer une relation inverse lorsque plusieurs groupes sont combinés) s'appelle le paradoxe de Simpson, et je vous encourage à consulter la page wikipédia qui y est consacrée.

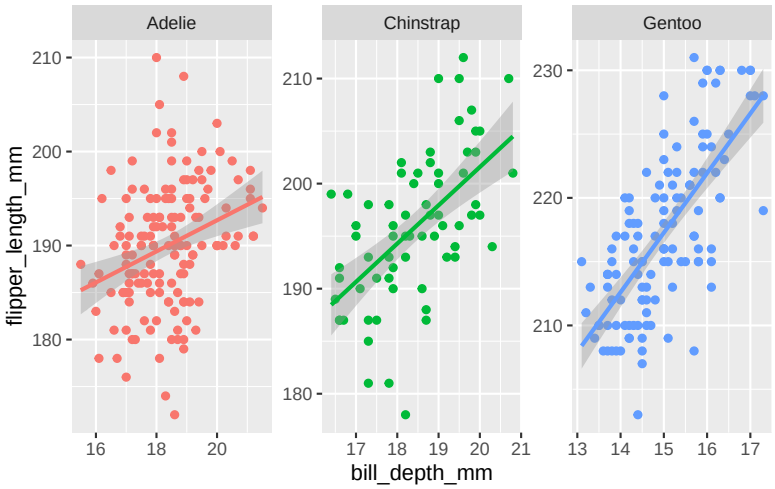
Ici, si la relation entre nos deux variables numériques s'inverse lorsque l'on examine cette relation à l'échelle de chaque modalité de la variable catégorielle. Une façon encore plus nette de mettre la relation positive en évidence est la suivante :

```
ggplot(penguins, aes(x = bill_depth_mm, y = flipper_length_mm,
                     color = species)) +
  geom_point(show.legend = FALSE) +
  geom_smooth(method = "lm", show.legend = FALSE) +
  facet_wrap(~species, scales = "free")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

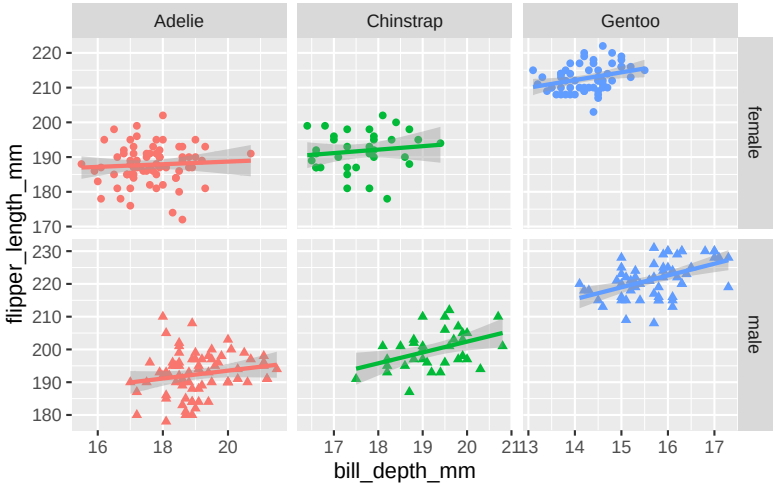


Même sans connaître en détail le principe et les limites de la régression linéaire, vous comprenez j'espère à quel point l'exploration rigoureuse d'un jeu de données est importante. Par exemple, nous avons vu que plus tôt que, pour chaque

espèce, les mâles sont plus lourds que les femelles. Est-ce que des différences morphologiques entre les sexes pourraient expliquer la relation que nous observons ici entre épaisseur du bec et longueur des nageoires ? Est-il possible qu'un second effet Simpson se cache dans ces données ? Si on distingue les deux sexes au sein de chaque espèce, la relation existe-t-elle toujours ? Et si oui, est-elle toujours positive ou s'inverse-t-elle à nouveau ? Pour le savoir, on peut utiliser une autre fonction permettant de produire des sous-graphiques, la fonction `facte_grid()`, qui permet de faire un sous graphique pour chaque combinaison de modalités de 2 variables catégorielles :

```
penguins |>
  filter(!is.na(sex)) |>    # Elimine les individus dont le sexe est inconnu
  ggplot(aes(x = bill_depth_mm, y = flipper_length_mm,
                     color = species, shape = sex)) +
  geom_point(show.legend = FALSE) +
  geom_smooth(method = "lm", show.legend = FALSE) +
  facet_grid(sex ~ species, scales = "free")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



La syntaxe `sex ~ species` indique que l'on souhaite un sous-graphique pour chaque combinaison des modalités des facteurs `sex` et `species`. Les graphiques correspondant aux différents sexes apparaîtront sur des lignes distinctes, et les espèces sur des colonnes distinctes. On constate ici que si la relation semble toujours positive et assez nette pour les mâles des

3 espèces, la situation est moins tranchée pour les femelles, en particulier pour les espèces Adélie et Chinstrap.

Nous avons vu dans ce chapitre quelques exemples et des règles à suivre strictement (notamment, quels types de graphiques pour quelles types de variables). Mais les possibilités sont infinies, et je vous encourage donc à poursuivre l'exploration. Toutes les combinaisons des éléments que nous avons décrits sont possibles. Entre les **facets**, qui permettent de faire des sous graphiques pour chaque modalités (ou combinaisons de modalités) d'une ou deux variables catégorielles, les **caractéristiques esthétiques** auxquelles ont peut associer un nombre conséquent de variables numériques et/ou catégorielles, et les nombreux **objets géométriques** existants (nous n'avons fait qu'utiliser les plus courants, mais il en existe *beaucoup d'autres*), les possibilités sont infinies. À vous de faire preuve de curiosité et d'explorer d'autres types de visualisation. L'avantage de `ggplot2` est que tous les graphiques se construisent sur le même modèle :

! Important

```
ggplot(TABLEAU, aes(x = VAR1, y = VAR2, fill = VAR3, ...)) +
  geom_XXX() +
  geom_YYY() +
  facet_ZZZ()
```

Quand on a bien compris ce principe, on peut quasiment tout faire, les réponses aux questions qu'on se pose se trouvant presque toujours dans les fichiers d'aide des fonctions.

3.11 Peaufiner l'apparence

Jusqu'ici, les morceaux de code que nous avons vus permettent de produire une large gamme de **graphiques exploratoires**. Mais il y a une différence de taille entre des graphiques que l'on fait pour soi, afin de comprendre et explorer des données, et des graphiques que l'on fait pour communiquer à autrui des informations ou le fruit de nos découvertes.

Les graphiques que l'on souhaite intégrer à un compte-rendu ou un rapport doivent :

- avoir des labels corrects pour les axes (penser à toujours indiquer l'unité des variables portées par les axes)
- avoir des labels corrects pour les légendes situées à droite de la plupart des graphiques (voir les nombreux exemples décrits plus haut)
- avoir éventuellement un titre. Il ne sera pas toujours utile de l'intégrer à la figure car la plupart du temps, les titres sont ajoutés manuellement dans le traitement de texte que vous utilisez
- utiliser des couleurs agréables et faciles à distinguer (y compris pour les personnes atteintes de daltonisme)
- utiliser des échelles adaptées (ordres de grandeur, échelles logarithmiques, etc.)
- si possible avoir tous le même thème (mêmes choix de couleurs, de contours, de polices de caractères, etc.)

Donc, lorsqu'on obtient un graphique exploratoire parlant et que l'on souhaite l'intégrer à un rapport ou un compte-rendu, 3 étapes sont nécessaires à sa mise en forme :

1. modifier les légendes avec la fonction `labs()`
2. modifier les échelles avec les nombreuses fonctions `scale_XXX_YYY()`
3. modifier le thème général avec les fonctions `theme_XX()`

3.11.1 Les légendes ou labels

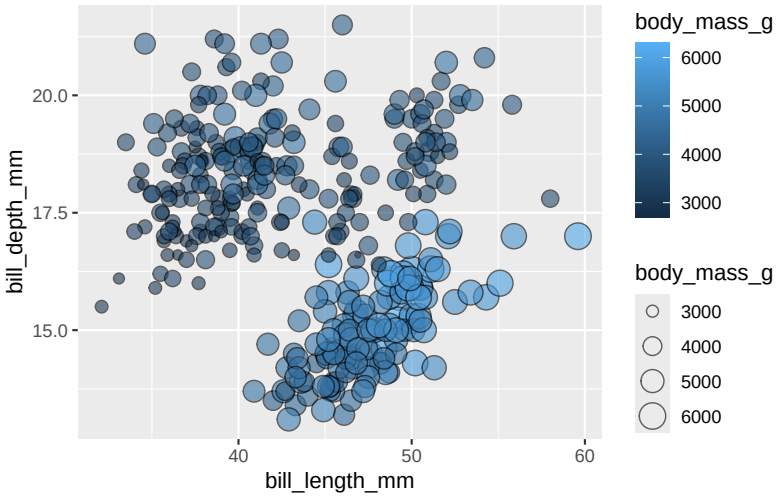
Le point de départ le plus évident est d'ajouter des labels de qualité. La fonction `labs()` du package `ggplot2` permet d'ajouter plusieurs types de labels sur vos graphiques :

- Un titre (`title =`) : il doit résumer les résultats les plus importants.
- Un sous-titre (`subtitle =`) : il permet de donner quelques détails supplémentaires.
- Une légende (`caption =`) : souvent utilisée pour présenter la source des données du graphique.
- Un titre pour chaque axe (`x =` et `y =`) : permet de préciser les variables portées par les axes et leurs unités.
- Un titre pour les échelles de couleurs, de forme, de taille, etc.

Reprenons par exemple le graphique permettant de visualiser la relation entre épaisseur et longueur du bec selon la masse des individus :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6)
```

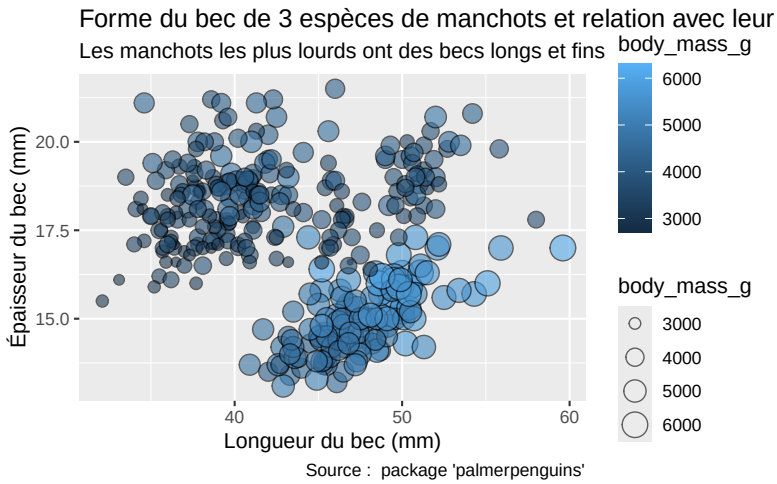
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Nous préciser les légendes en ajoutant la fonction `labs()` sur une nouvelle couche du graphique :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'")
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Pour annoter correctement les légendes situées à droite, il convient d'avoir bien compris ce qui, dans notre code, a permis à `R` de générer automatiquement ces légendes. Le gradient de couleur a été créé parce que nous avons tapé `fill = body_mass_g`. Et l'échelle de taille des symboles a été créée parce que nous avons tapé `size = body_mass_g`. Dans la fonction `labs()`, nous devons donc préciser `fill = "..."` et `size = "..."` pour modifier le titre de ces 2 légendes :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)")
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

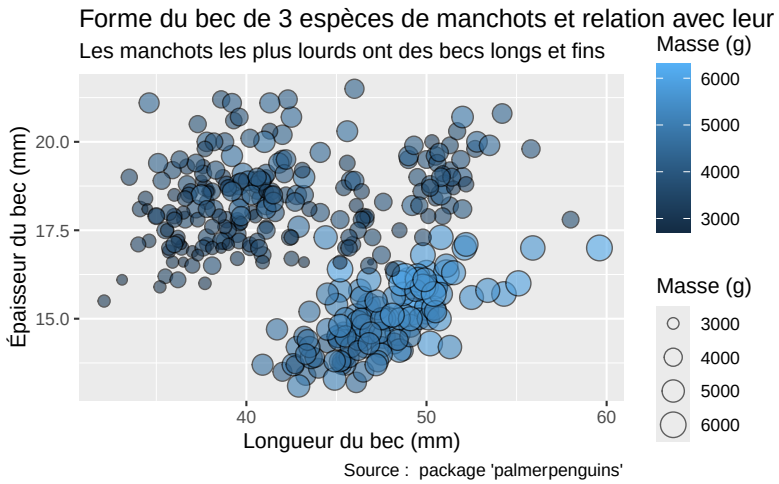


Figure 3.18: Une figure correctement légendée

À partir de maintenant, vous devriez systématiquement légender les axes de vos graphiques et annoter vos légendes correctement, en n'oubliant pas de préciser les unités lorsque c'est pertinent, pour tous les graphiques que vous intégrez dans vos rapports, compte-rendus, mémoires, présentation, etc.

3.11.2 Les échelles

Tous les détails des graphiques que vous produisez peuvent être édités. C'est notamment le cas des échelles. Qu'il s'agisse de modifier l'étendue des axes, la densité du quadrillage, la position des tirets sur les axes, le nom des catégories figurant sur les axes ou dans les légendes ou encore les couleurs utilisées pour différentes catégories d'objets géométriques, tout est possible dans `ggplot2`.

Nous n'avons pas le temps ici d'aborder toutes ces questions en détail. Je vous encourage donc à consulter l'ouvrage en ligne intitulé R for data science, et en particulier son chapitre dédié aux échelles, si vous avez besoin d'apporter des modifications à vos graphiques et que vous ne trouvez pas comment faire dans cet ouvrage.

3.11.2.1 La gestion des couleurs

Nous allons néanmoins examiner quelques possibilités, à commencer par la façon de procéder pour modifier les couleurs choisies par défaut par `ggplot2`. Reprenons la figure Figure 3.18, et changeons le gradient de couleur proposé par défaut par `R`. Il est possible de modifier ces couleurs de plusieurs façons :

- soit en utilisant d'autres palettes de couleurs prédéfinies
- soit en choisissant manuellement les couleurs

Toutes les fonctions permettant d'altérer les légendes commencent par `scale_`. Vient ensuite le nom de l'esthétique que l'on souhaite modifier (ici `fill_`) et enfin, le nom d'une fonction à appliquer. Les possibilités sont nombreuses et vous pouvez en avoir un aperçu en tapant le début du nom de la fonction et en parcourant la liste proposée par `RStudio` sous le curseur. Il faut toutefois distinguer 2 types d'échelles de couleurs : les échelles continues (c'est notre cas ici) et les échelles discrètes (quand l'esthétique de couleur est associée à une variable catégorielle, nous en verrons un exemple plus loin).

Par exemple, il est possible d'utiliser la palette `viridis`. Selon ses auteurs :

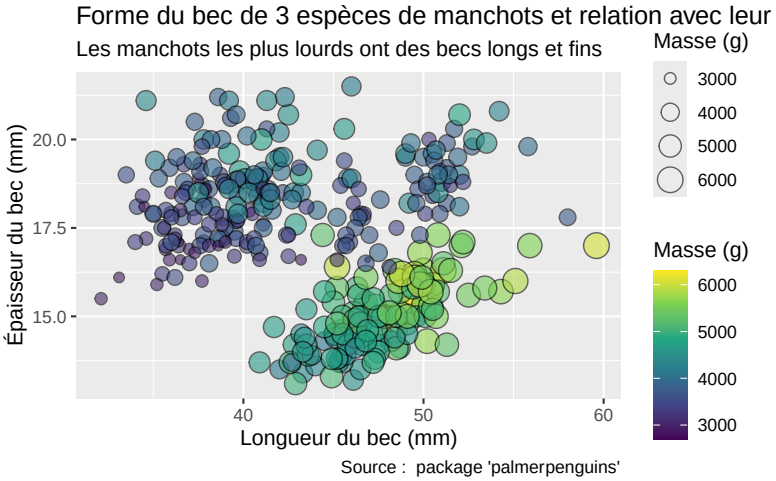
> "Use [this palette] to make plots that are pretty, better represent your data, easier to read by those with colorblindness, and print well in gray scale.

Pour utiliser cette palette, il suffit d'ajouter une couche à notre graphique :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
```

```
    scale_fill_viridis_c()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

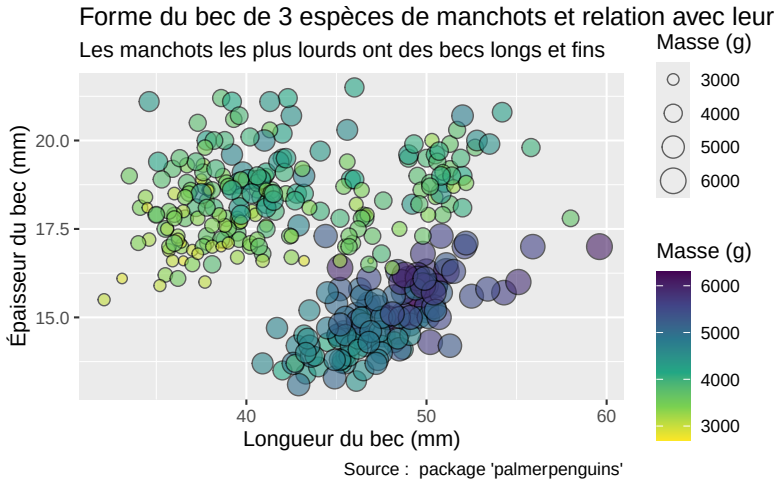


La palette `viridis` est proposée pour les échelles continues (d'où le `_c` à la fin du nom de fonction), ou pour les échelles discrètes (`scale_fill_viridis_d`). Des fonctions équivalentes existent pour les couleurs de contour (`scale_color_viridis_c` et `scale_color_viridis_d`). Allez lire le fichier d'aide de cette fonction pour en apprendre plus sur son fonctionnement et ses nombreuses options.

Ici, les individus les plus lourds apparaissent en jaune, et les plus légers en bleu sombre. Si on souhaite faire le contraire, c'est possible :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
  scale_fill_viridis_c(direction = -1)
```

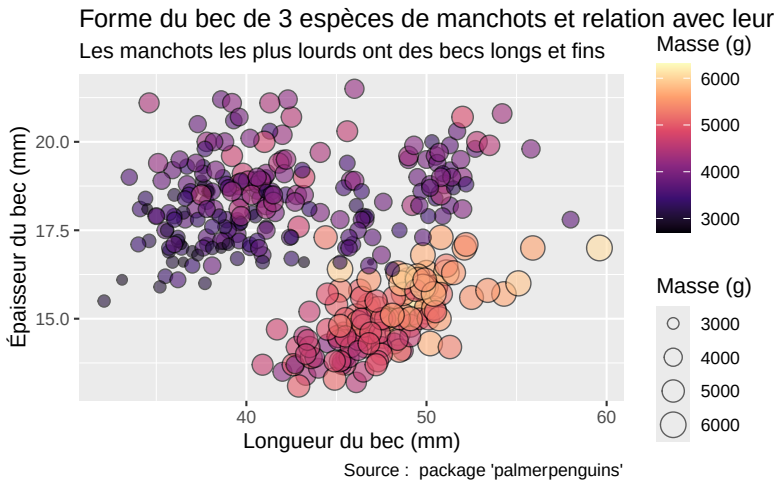
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



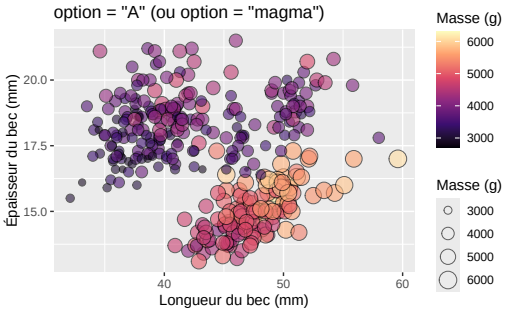
D'autres palettes de couleurs sont également accessibles grâce à l'argument `option =` de ces fonctions :

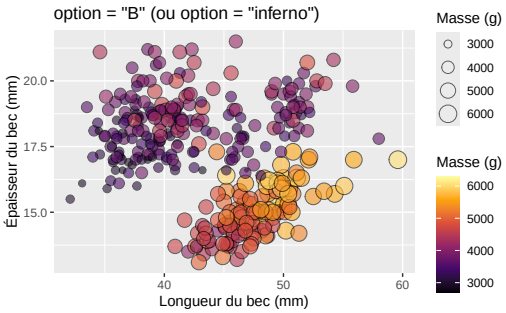
```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
  scale_fill_viridis_c(option = "A")
```

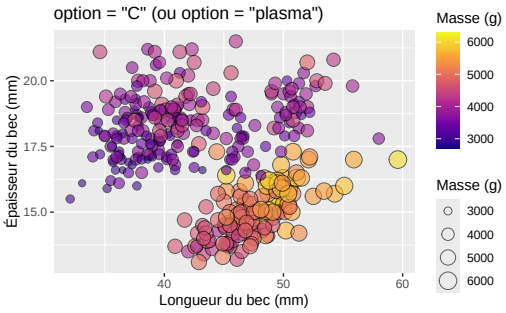
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

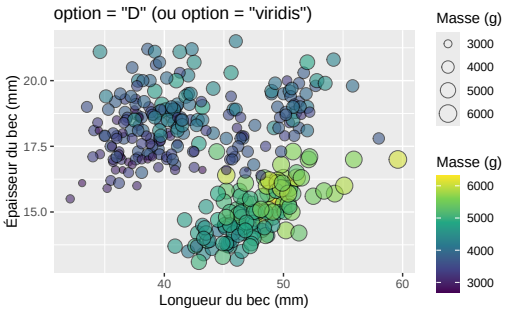


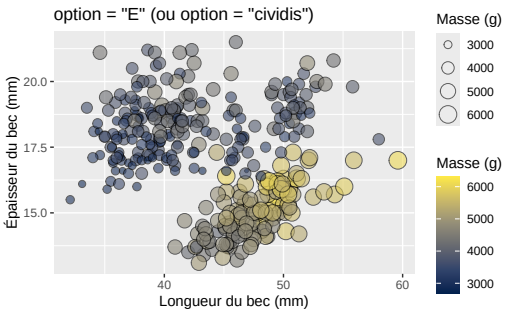
Voici toutes les possibilités :


option = "A" (ou option = "magma")
Épaisseur du bec (mm)
20.0
17.5
15.0
40
50
60
Longueur du bec (mm)
Masse (g)
6000
5000
4000
3000
Masse (g)
3000
4000
5000
6000


option = "B" (ou option = "inferno")
Épaisseur du bec (mm)
20.0
17.5
15.0
40
50
60
Longueur du bec (mm)
Masse (g)
3000
4000
5000
6000
Masse (g)
6000
5000
4000
3000


option = "C" (ou option = "plasma")
Épaisseur du bec (mm)
20.0
17.5
15.0
40
50
60
Longueur du bec (mm)
Masse (g)
6000
5000
4000
3000
Masse (g)
3000
4000
5000
6000

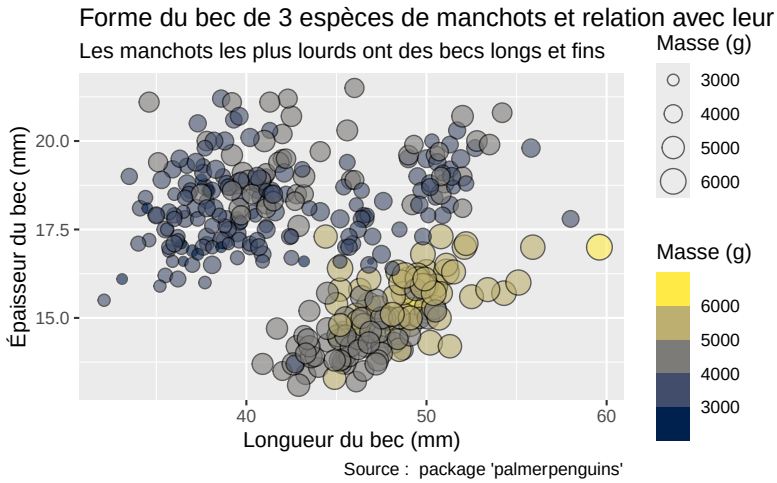

option = "D" (ou option = "viridis")
Épaisseur du bec (mm)
20.0
17.5
15.0
40
50
60
Longueur du bec (mm)
Masse (g)
3000
4000
5000
6000
Masse (g)
6000
5000
4000
3000


option = "E" (ou option = "cividis")
Épaisseur du bec (mm)
20.0
17.5
15.0
40
50
60
Longueur du bec (mm)
Masse (g)
3000
4000
5000
6000
Masse (g)
6000
5000
4000
3000

Dernière chose concernant `viridis` : la fonction `scale_fill_viridis_b()` discrétise la variable continue pour en faire une échelle discontinue. Il est en effet parfois plus facile de repérer une couleur parmi une palette de 4 ou 5 couleurs distinctes, plutôt qu'au sein d'un gradient. Voilà à quoi cela ressemble :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
  scale_fill_viridis_b(option = "E")
```

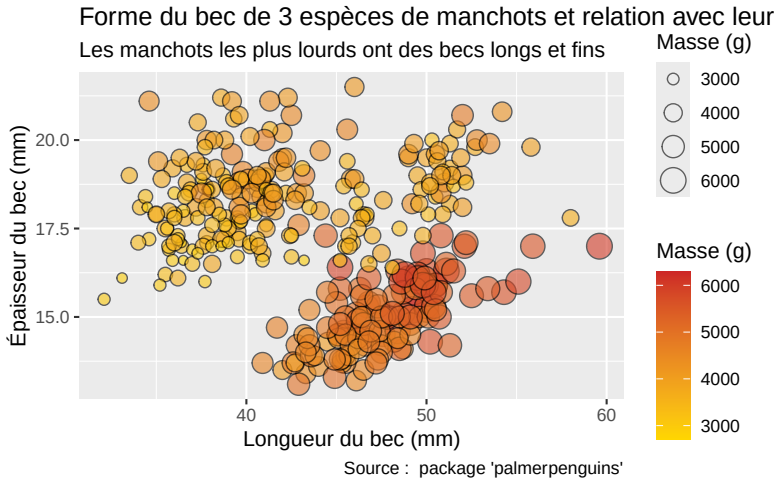
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Outre les fonctions d'échelles proposant les palettes `viridis`, les fonctions se terminant par `_gradient()`, `_gradient2()` et `_gradientn()` permettent de spécifier manuellement les couleurs à intégrer dans un dégradé. Avec la fonction `scale_fill_gradient()` on indique simplement les couleurs du début de de la fin du gradient :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
  scale_fill_gradient(low = "gold", high = "firebrick3")
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

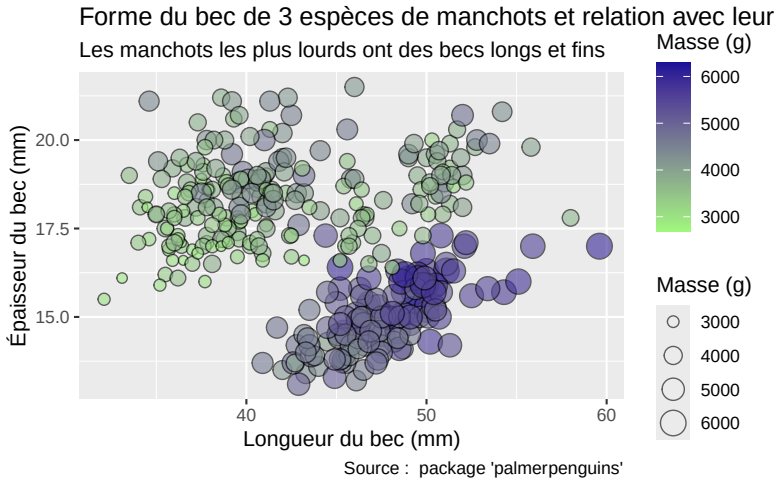


N'importe quelle nom de couleur valide, ou n'importe que code couleur hexadécimal fonctionne (voir par exemple ce site pour trouver les codes hexadécimaux dont vous avez besoin) :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
```

```
      fill = "Masse (g)",
      size = "Masse (g)") +
  scale_fill_gradient(low = "#A0F87D", high = "#151197")
```

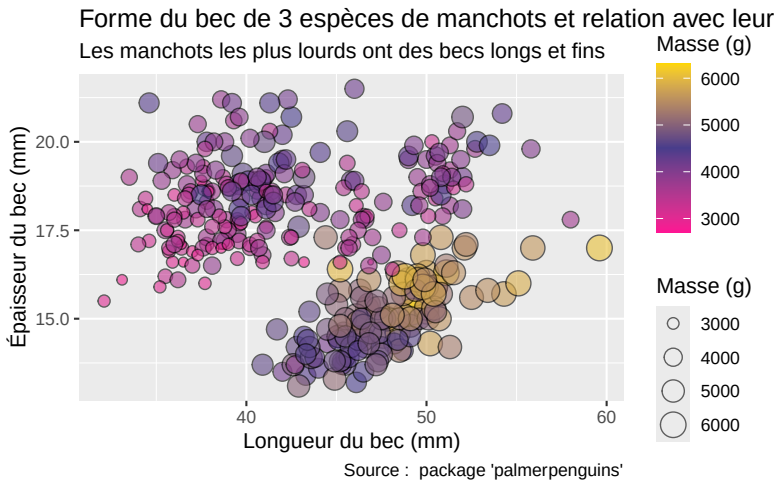
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Avec la fonction `scale_fill_gradient2()`, nous avons plus de contrôle : on indique une couleur de départ, une couleur d'arrivée, mais aussi, une couleur intermédiaire, et la valeur numérique à laquelle cette couleur doit apparaître :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = body_mass_g, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Masse (g)",
       size = "Masse (g)") +
  scale_fill_gradient2(low = "deeppink", high = "gold", mid = "darkslateblue",
                       midpoint = 4500)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

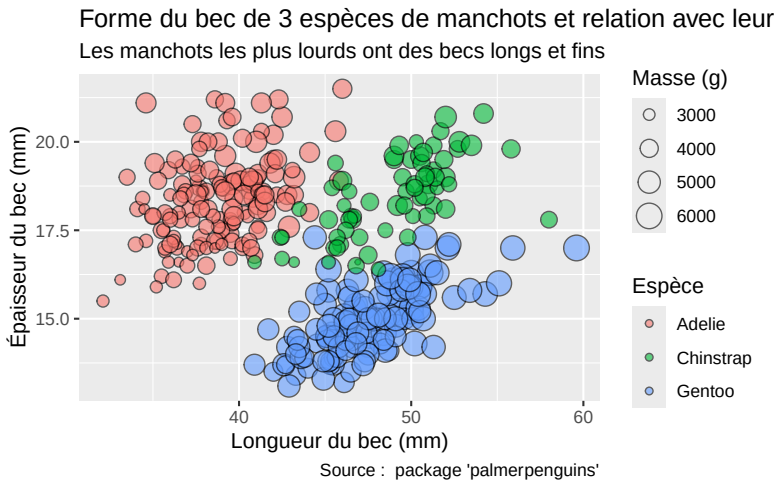


Je vous laisse explorer l'aide de cette fonction ainsi que celle de `scale_fill_gradientn()` pour savoir comment en utiliser toutes les possibilités.

Lorsqu'une variable catégorielle est associée à la couleur, il est évidemment possible aussi d'effectuer les choix de palettes. Voyons en exemple en associant la couleur de remplissage des points à l'espèce plutôt qu'à la masse :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)")
```

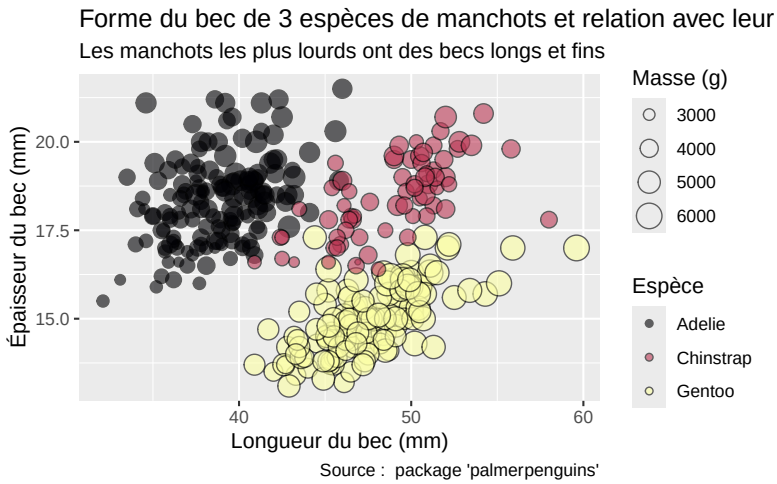
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



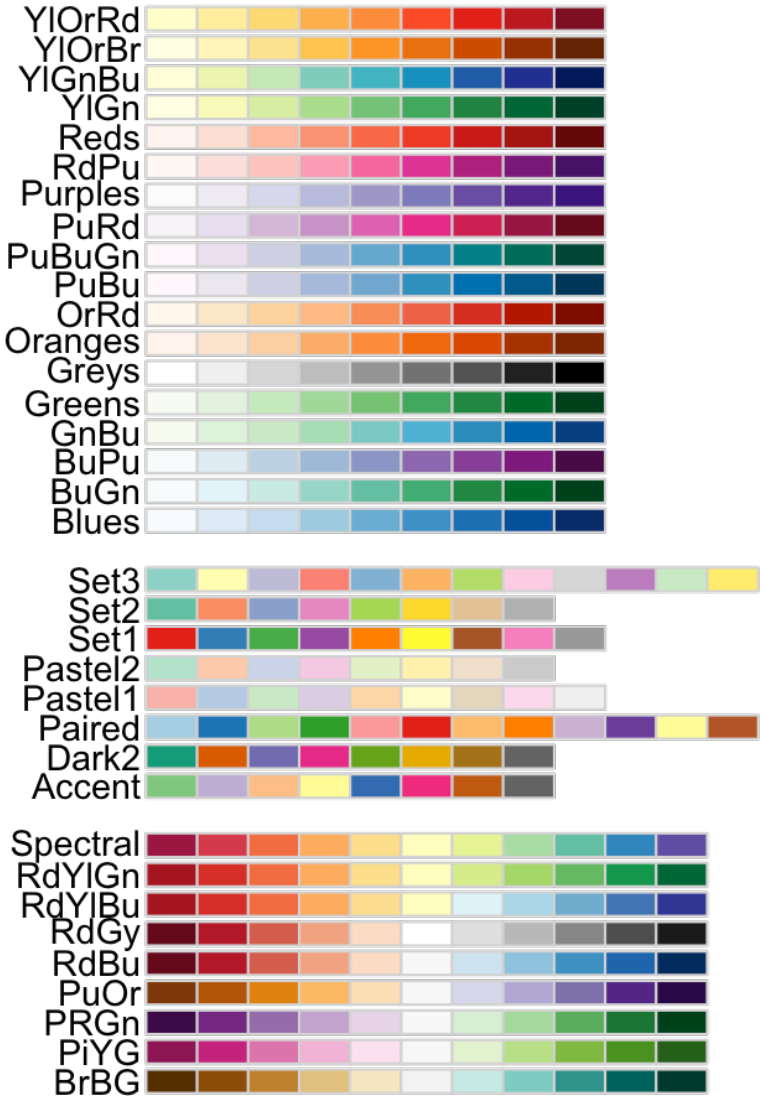
La version discrète de `viridis` peut maintenant être appliquée. Les mêmes options que pour la version continue sont disponibles :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B")
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Les possibilités sont nombreuses, notamment grâce aux fonctions `scale_fill_brewer()` (pour les couleurs de remplissages associées à une variable catégorielle) et `scale_color_brewer()` (pour les couleurs de contour associées à une variable catégorielle). L'utilisation est simple, un précise simplement quelle palette on souhaite utiliser parmi la liste des palettes disponibles :

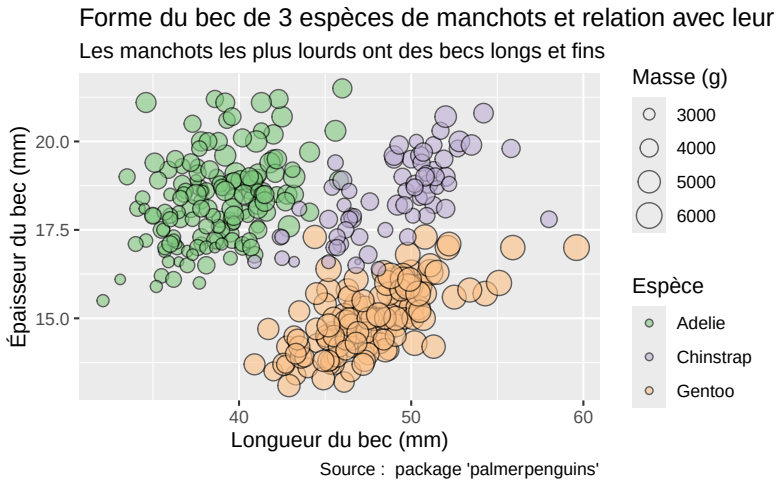


Certaines palettes sont séquentielles (lorsque les catégories se suivent logiquement, pour les variables catégorielles ordinales en particulier), d'autres contiennent des couleurs indépendantes :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
```

```
      y = "Épaisseur du bec (mm)",
      caption = "Source :  package 'palmerpenguins'",
      fill = "Espèce",
      size = "Masse (g)") +
  scale_fill_brewer(palette = "Accent")
```

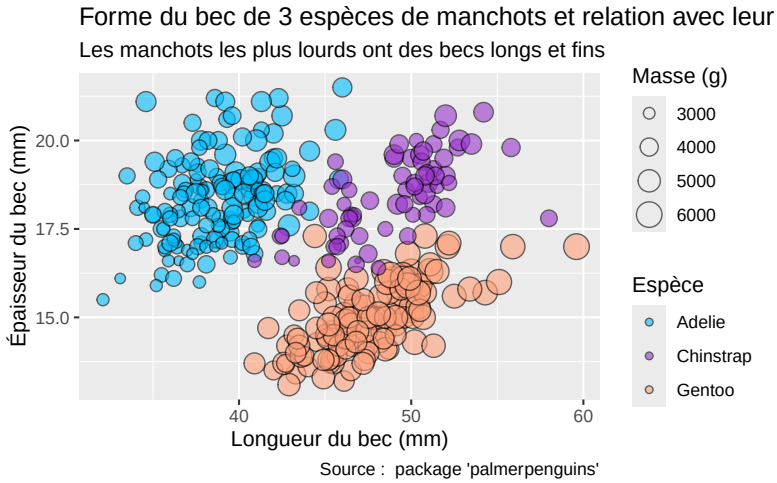
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Enfin, il est possible de spécifier manuellement la liste des couleurs que l'on souhaite utiliser avec la fonction `scale_fill_manual()`. il faut bien entendu indiquer autant de couleurs que de modalités pour notre variable catégorielle (ici 3 espèces dont 3 couleurs) :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
      subtitle = "Les manchots les plus lourds ont des becs longs et fins",
      x = "Longueur du bec (mm)",
      y = "Épaisseur du bec (mm)",
      caption = "Source :  package 'palmerpenguins'",
      fill = "Espèce",
      size = "Masse (g)") +
  scale_fill_manual(values = c("deepskyblue", "darkorchid3", "lightsalmon"))
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



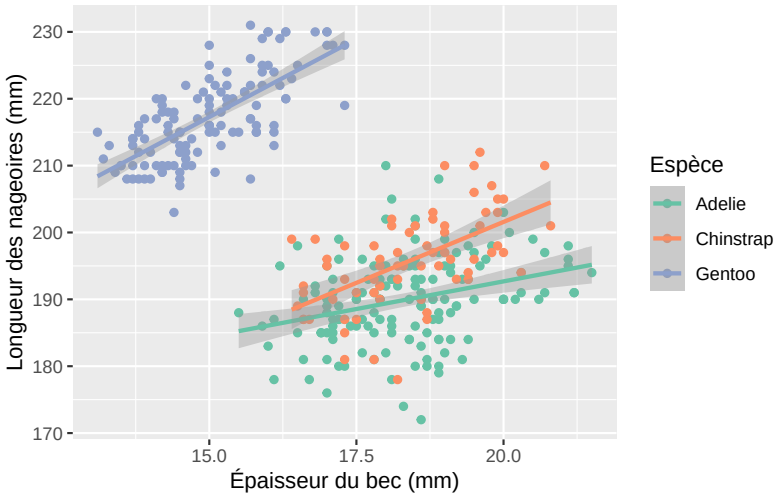
Dernière chose concernant les couleurs : un choix de fonction `scale_XXX_XXX()` inapproprié est une cause d'erreur très fréquente ! Par exemple, pour la première figure de la partie consacrée au paradoxe de Simpson (Section 3.10.3), la couleur des points et des lignes n'est pas spécifiée avec `fill` mais avec `color`. C'est donc bien une fonction qui commence par `scale_color_` qu'il faut utiliser :

```
ggplot(penguins, aes(x = bill_depth_mm, y = flipper_length_mm,
                     color = species)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "Épaisseur du bec (mm)", y = "Longueur des nageoires (mm)",
       color = "Espèce") +
  scale_color_brewer(palette = "Set2")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

```
Warning: Removed 2 rows containing non-finite outside the scale range
(`stat_smooth()`).
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Comme pour les fonctions `geom_XXX()`, les fonctions `scale_color_XXX()` et `scale_fill_XXX()` sont très nombreuses. Je vous encourage donc à explorer les fichiers d'aide et à faire des essais.

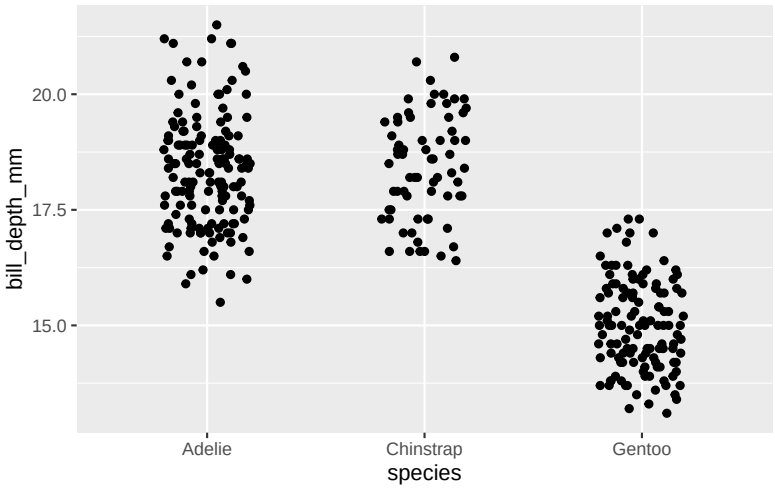
3.11.2.2 Les autres échelles

Les deux autres échelles que vous pourrez être couramment appelés à modifier sont les échelles des axes des `x` et des `y`. Les fonctions qui permettent de la faire sont construites comme ces des échelles de couleurs : `scale_x_XXX()` et `scale_y_XXX()`. La dernière partie du nom de la fonction sera, la plupart du temps, soit `discrete` si une variable catégorielle est associée à l'axe, soit `continuous` si une variable numérique y est associée.

Reprenons l'exemple du stripchart suivant :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0)
```

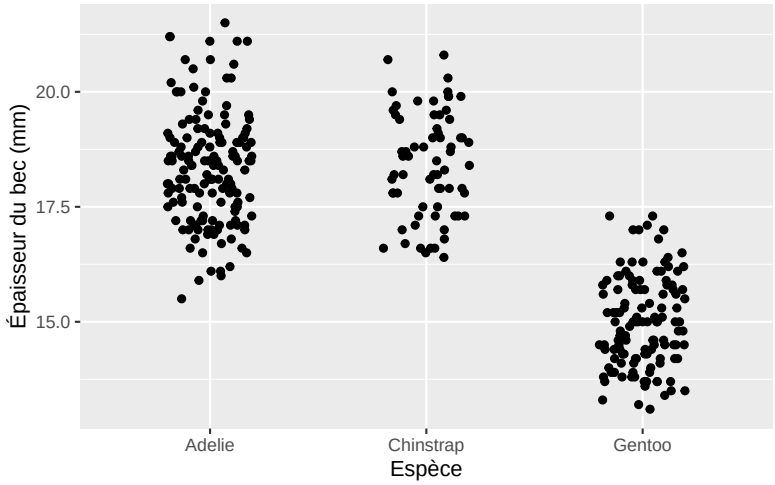
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



On commence par légender les axes avec `labs()` :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0) +
  labs(x = "Espèce", y = "Épaisseur du bec (mm)")
```

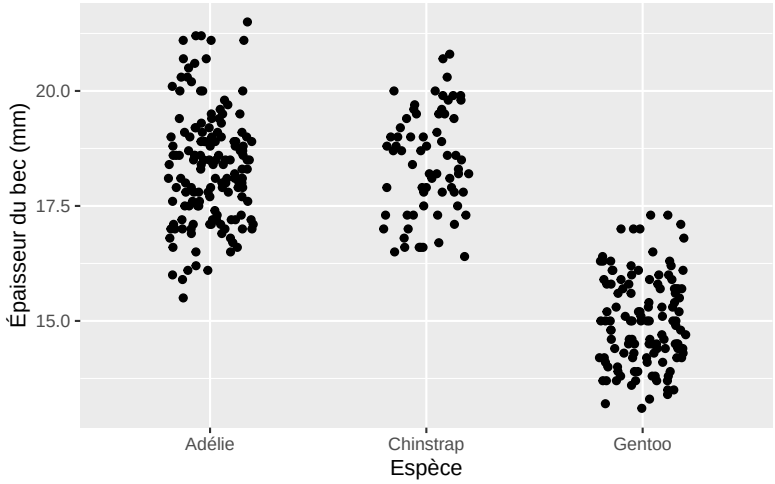
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Si on souhaite renommer l'espèce `Adelie` en `Adélie` (avec un accent sur le "e" donc), il faut modifier l'échelle de l'axe des `x`, qui porte une variable catégorielle :

```r
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0) +
  labs(x = "Espèce", y = "Épaisseur du bec (mm)") +
  scale_x_discrete(label = c("Adélie", "Chinstrap", "Gentoo"))
```

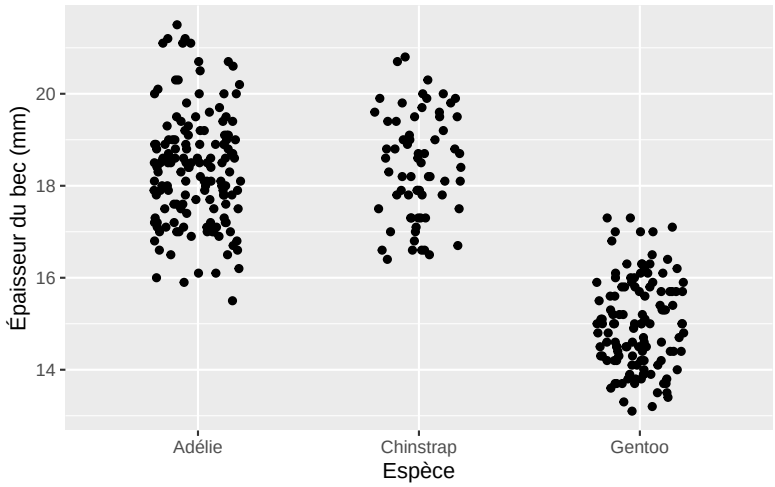
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Pour l'axe des y, qui porte une variable continue, on peut avoir besoin de faire apparaître des graduations tous les 2 millimètres (au lieu de tous les 2,5 millimètres) :

```r
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0) +
  labs(x = "Espèce", y = "Épaisseur du bec (mm)") +
  scale_x_discrete(label = c("Adélie", "Chinstrap", "Gentoo")) +
  scale_y_continuous(breaks = seq(from = 12, to = 22, by = 2))
```

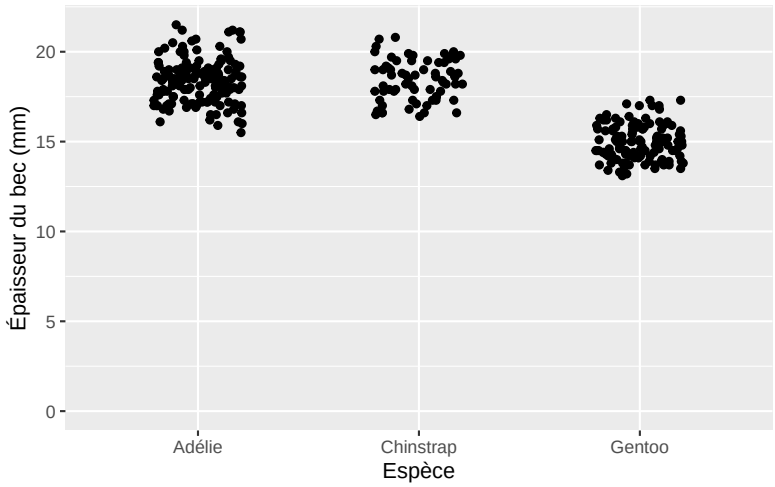
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Il est également très fréquent de souhaiter étendre les axes au-delà des seules valeurs observées, pour faire apparaître le 0 par exemple. C'est tellement fréquent qu'une fonction de raccourci très facile à utiliser nous permet d'éviter le recours à une fonction `scale_XXX_XXX()`. Même si dans ce cas précis, ça n'est pas très pertinent, voilà un exemple :

```
ggplot(penguins, aes(x = species, y = bill_depth_mm)) +
  geom_jitter(width = 0.20, height = 0) +
  labs(x = "Espèce", y = "Épaisseur du bec (mm)") +
  scale_x_discrete(label = c("Adélie", "Chinstrap", "Gentoo")) +
  expand_limits(y = 0)
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Enfin, il arrive que les valeurs prises par une variable numérique recouvrent plusieurs ordres de grandeurs (avec par exemple des valeurs de l'ordre des dizaines, des centaines et des milliers). Utiliser une échelle logarithmique permet, dans cette situation, de mieux visualiser la variabilité des données, notamment parmi les valeurs les plus faibles. Les fonctions `scale_x_log10()` et `scale_y_log10()` permettent d'effectuer ce changement d'échelle tout en conservant des valeurs normales sur les axes.

Outre ces changements d'échelles pour les axes et les couleurs, il est possible de modifier manuellement toutes les échelles générées automatiquement par les fonction `geom_XXX()` (par exemple, l'échelle des tailles, ou les types de symboles utilisés pour distinguer plusieurs catégories de points). Il "suffit" pour cela de trouver la bonne fonction (par exemple `scale_size_continuous()`, `scale_shape_manual()`, … Il est evidemment impossible de faire le tour de toutes ces fonctions. Mais sachez qu'elles existent et consultez leurs fichiers d'aide le jour où vous en avez besoin.

3.11.3 Les thèmes

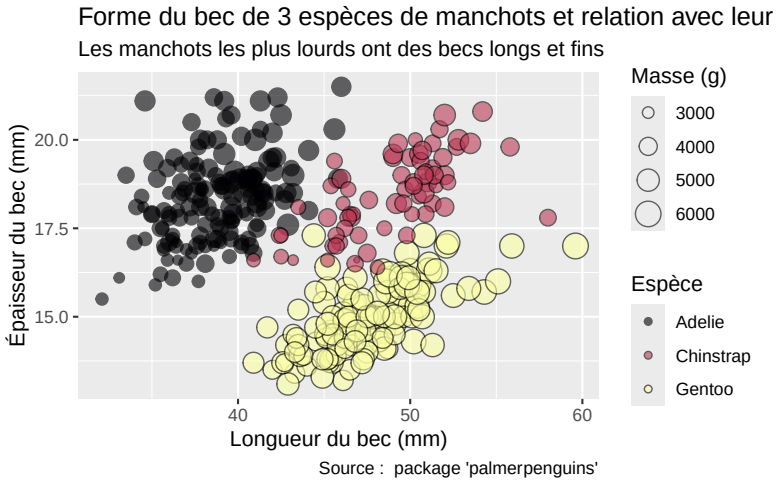
L'apparence de tout ce qui ne concerne pas directement les données d'un graphique est sous le contrôle d'un thème. Les thèmes contrôlent l'apparence générale du graphique : quelles polices et tailles de caractères sont utilisées, quel sera l'arrière plan du graphique, faut-il intégrer un quadrillage sous le graphique, et si oui, quelles doivent être ses caractéristiques ?

Il est possible de spécifier chaque élément manuellement. Nous nous contenterons ici de passer en revue quelques thèmes prédéfinis qui devraient couvrir la plupart de vos besoins.

Reprenons par exemple le code suivant :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B")
```

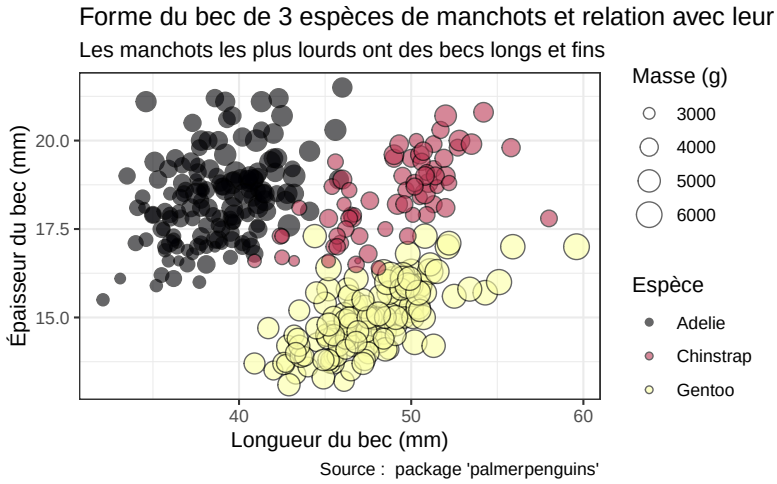
```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



Le thème utilisé par défaut est `theme_gray()`. Il est notamment responsable de l'arrière plan gris et du quadrillage blanc. Pour changer de thème, il suffit d'ajouter une couche au graphique en donnant le nom du nouveau thème :

```r
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B") +
  theme_bw()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```



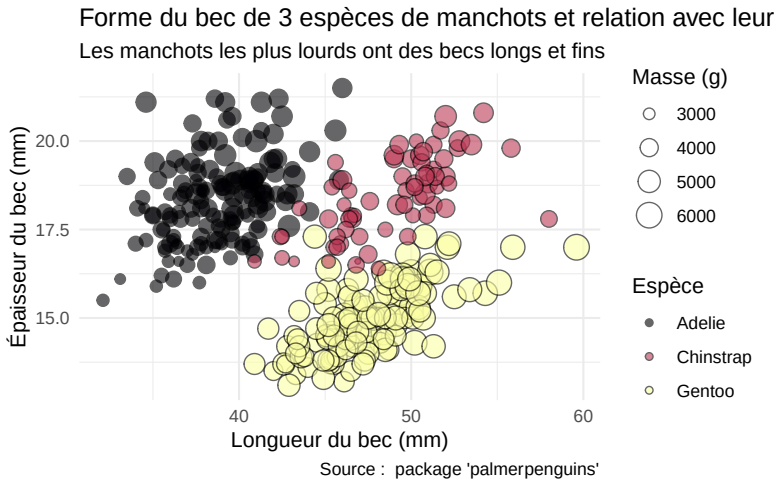
Le fond gris a disparu, et le quadrillage a changé de couleur. Les thèmes complets proposés par `ggplot2` que vous pouvez utiliser sont les suivants :

- `theme_bw()` : fond blanc et quadrillage.
- `theme_classic()` : thème classique, avec des axes mais pas de quadrillage.
- `theme_dark()` : fond sombre pour augmenter le contraste.
- `theme_gray()` : thème par défaut : fond gris et quadrillage blanc.
- `theme_light()` : axes et quadrillages discrets.

- `theme_linedraw()` : uniquement des lignes noires.
- `theme_minimal()` : pas d'arrière plan, pas d'axes, quadrillage discret.
- `theme_void()` : thème vide, seuls les objets géométriques restent visibles.

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B") +
  theme_minimal()
```

```
Warning: Removed 2 rows containing missing values or values outside the scale range
(`geom_point()`).
```

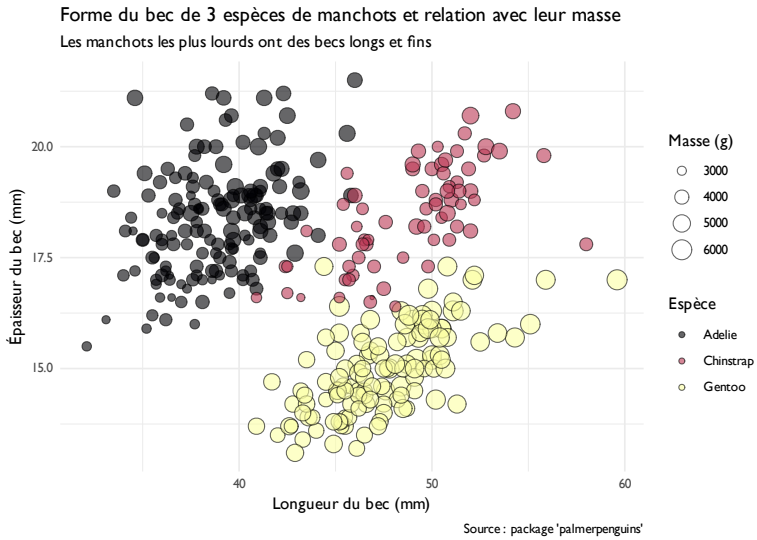


Tous les thèmes possèdent la même liste d'argument. L'un d'entre eux est l'argument `base_family`, qui permet de spécifier une police de caractères différente de celle utilisée par défaut. Évidemment, vous ne pourrez utiliser que des polices qui sont disponibles sur l'ordinateur que vous utilisez. Un bon tutoriel expliquant comment indiquer à `R` les polices qui sont

disponibles sur votre ordinateur est disponible ici. N'hésitez pas à revenir vers moi pour toute question à ce sujet.

Dans l'exemple ci-dessous, j'utilise la police "Gill Sans". Si cette police n'est pas disponible sur votre ordinateur, ce code produira une erreur (ou `R` prendra simplement la police par défaut. Si c'est le cas, remplacez-la par une police de votre ordinateur. Attention, son nom exact doit être utilisé. Cela signifie bien sûr le respect des espaces, majuscules, etc.

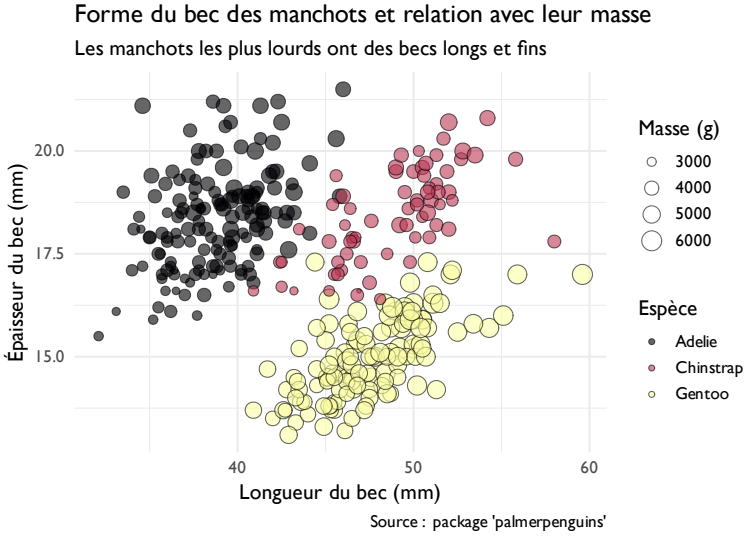
```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec de 3 espèces de manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B") +
  theme_minimal(base_family = "Gill Sans")
```



Il est également possible de spécifier la taille de police qui devrait être utilisée par défaut. On spécifie la taille de base avec l'argument `base_size =`, et toutes les autres tailles de polices seront mises à jour pour refléter le changement. Ainsi,

les différences de tailles entre titre, sous-titres, légendes des axes, etc, seront maintenues :

```
ggplot(penguins, aes(x = bill_length_mm, y = bill_depth_mm,
                     fill = species, size = body_mass_g)) +
  geom_point(shape = 21, color = "black", alpha = 0.6) +
  labs(title = "Forme du bec des manchots et relation avec leur masse",
       subtitle = "Les manchots les plus lourds ont des becs longs et fins",
       x = "Longueur du bec (mm)",
       y = "Épaisseur du bec (mm)",
       caption = "Source :  package 'palmerpenguins'",
       fill = "Espèce",
       size = "Masse (g)") +
  scale_fill_viridis_d(option = "B") +
  theme_minimal(base_family = "Gill Sans", base_size = 14)
```



Le choix d'un thème et d'une police adaptés doivent vous permettre de faire des graphiques originaux et clairs. Rappelez-vous toujours que vos choix en matière de graphiques doivent avoir pour objectif principal de rendre les tendances plus faciles à décrypter pour un lecteur non familier de vos données. C'est un outil de communication au même titre que n'importe quel paragraphe d'un rapport ou compte-rendu. Et comme pour un paragraphe, la première version d'un graphique est rarement la bonne.

Vous devriez donc maintenant être bien armés pour produire 95% des graphiques dont vous aurez besoin tout au long de

votre cursus universitaire. Toutefois, un point important a pour l'instant été omis : l'ajout de barres d'erreurs sur vos graphiques. Nous verrons comment faire cela au prochain semestre, après avoir appris à manipuler efficacement des tableaux de données avec les packages `tidyr` et `dplyr`.

Quoi qu'il en soit, il est maintenant attendu de vous que vous utilisez `R` et ce que vous avez appris de `ggplot2` pour produire tous les graphiques que vous serez amenés à intégrer à vos comptes-rendus de TP et à vos rapports.

3.12 Exercices

1. Avec le jeu de données `diamonds`, du packages `ggplot2`, tapez les commandes suivantes pour créer un nouveau tableau `diams` contenant moins de lignes (3000 au lieu de près de 54000) :

```
library(dplyr)
set.seed(4532) # Afin que tout le monde récupère les mêmes lignes
diams <- diamonds |>
  sample_n(3000)
```

2. Avec ce nouveau tableau `diams`, tapez le code permettant de créer le graphique ci-dessous. Indice : affichez le tableau `diams` dans la console afin de voir quelles sont les variables disponibles.

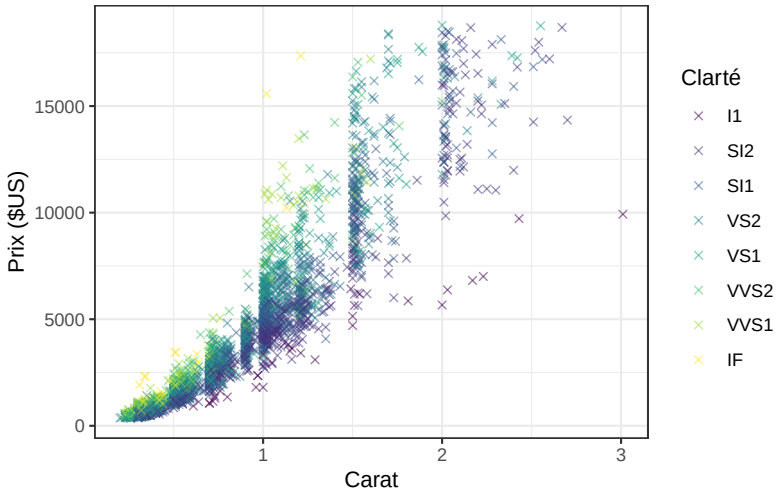


Figure 3.19: Prix de 3000 diamants en fonction de leur taille en carats et de leur clarté.

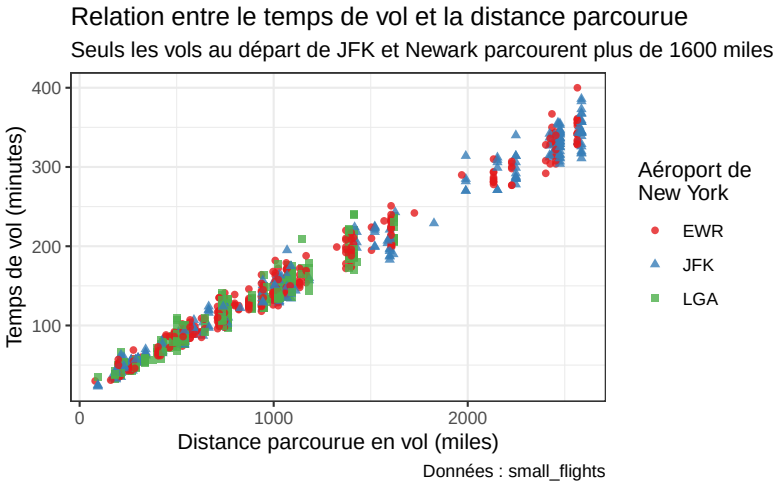
3. Selon vous, à quoi sont dues les bandes verticales que l'on observe sur ce graphique ?
4. Installez et chargez en mémoire le package `nycflights13`
5. Examinez le tableau `flights` de ce package, et lisez son fichier d'aide pour comprendre à quoi correspondent ces données
6. Créer un nouveau jeu de données en exécutant ces commandes :

```
set.seed(1234)
small_flights <- flights |>
  filter(!is.na(arr_delay),
         distance < 3000)  |>
  sample_n(1000)
```

7. Ce nouveau jeu de données de petite taille (1000 lignes) est nommé `small_flights`. Il contient les mêmes variables que le tableau `flights` mais ne contient qu'une petite fraction de ses lignes. Les lignes retenues ont été choisies au hasard. Vous pouvez visualiser son contenu en tapant son nom dans la console ou en utilisant la fonction `View()`.

En vous appuyant sur les fonctions et les principes de la grammaire des graphiques que vous avez découverts dans ce cha-

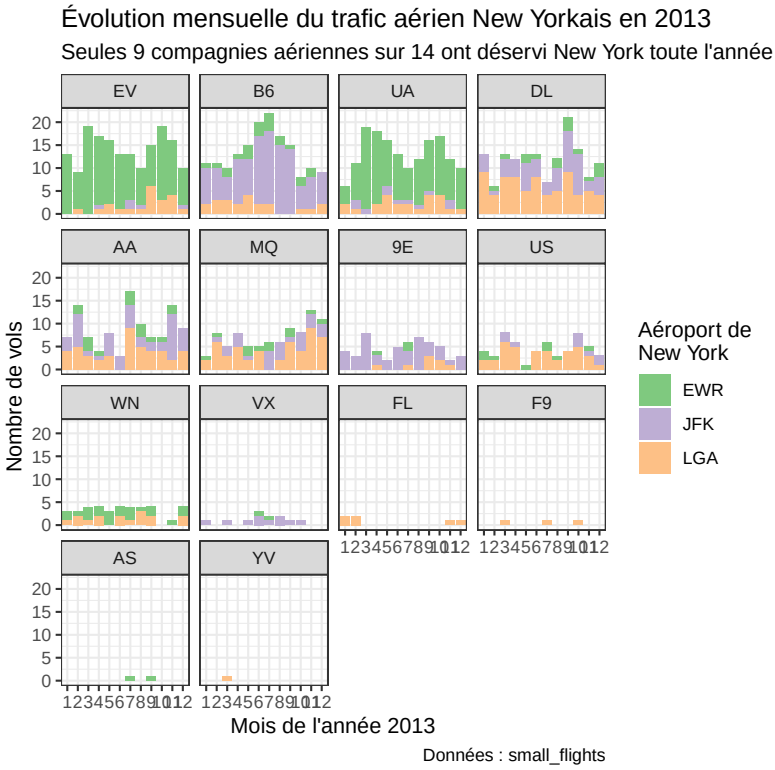
pitre, et en vous servant de ce nouveau jeu de données, tapez les commandes qui permettent de produire le graphique ci-dessous :



Quelques indices :

- Les couleurs utilisées sont celles de la palette `Set1` du package `RColorBrewer`.
- Les variables utilisées sont `origin`, `air_time` et `distance`.
- La transparence des symboles est fixée à `0.8`.

8. Toujours avec ce jeu de données `small-flights`, tapez les commandes permettant de produire le graphique ci-dessous :



Quelques indices :

- Les couleurs utilisées sont celles de la palettes `Accent` du package `RColorBrewer`.
- Les variables utilisées sont `month`, `carrier` et `origin`.
- La variable `month` est codée sous forme numérique dans le tableau de données. Il faudra la transformer en facteur avec la fonction `factor(month)` au moment de l'associer à un axe du graphique.

4 Manipuler des tableaux avec `dplyr`

4.1 Pré-requis

Nous abordons ici une étape essentielle de toute analyse de données : la manipulation de tableaux, la sélection de lignes, de colonnes, la création de nouvelles variables, etc. Bien souvent, les données brutes que nous importons dans `R` ne sont pas utiles en l'état. Il nous faut parfois sélectionner seulement certaines lignes pour travailler sur une petite partie du jeu de données. Il nous faut parfois modifier des variables existantes (pour modifier les unités par exemple) ou en créer de nouvelles à partir des variables existantes. Nous avons aussi très souvent besoin de constituer des groupes et d'obtenir des statistiques descriptives pour chaque groupe (moyenne, écart-type, erreur type, etc). Nous verrons dans ce chapitre comment faire tout cela grâce au package `dplyr` qui fournit un cadre cohérent et des fonctions simples permettant d'effectuer tous les tripatouillages de données dont nous pourrons avoir besoin.

Dans ce chapitre, nous aurons besoin des packages suivants :

```
library(dplyr)
library(ggplot2)
library(palmerpenguins)
library(readxl)
```

4.2 Importer des données depuis un tableur

4.2.1 Les règles de base

Jusqu'à maintenant, nous avons travaillé exclusivement avec des jeux de données déjà disponibles dans `R`. La plupart du temps, les données sur lesquelles vous devrez travailler devront au préalable être importées dans `R`, à partir de fichiers issus de tableurs. De tels fichiers se présentent généralement sous l'un des 2 formats suivants :

1. Fichiers au format ".csv" : il s'agit d'un format de fichier dit "texte brut", c'est à dire qu'il peut être ouvert avec n'importe quel éditeur de texte, y compris le bloc notes de Windows. L'extension ".csv" est l'abréviation de "Comma Separated Values", autrement dit, dans ce type de fichiers, les colonnes sont séparées par des virgules. Cela peut poser problème en France puisque le symbole des décimales est souvent aussi la virgule (et non le point comme dans les pays anglo-saxons). Le séparateur de colonnes utilisé en France dans les fichiers `.csv` est alors souvent le point-virgule. Il est possible de créer des fichiers `.csv` à partir de n'importe quel tableur en choisissant `Fichier > Exporter...` ou `Fichier > Enregistrer sous...` puis en sélectionnant le format approprié (les dénominations sont variables selon les logiciels : format texte brut, format csv, plain text, etc...).
2. Fichiers au format tableur : `.xls` ou `.xlsx` pour Excel, `.calc` pour Open Office.

Dans les 2 cas, pour que `R` puisse importer les données contenues dans ces fichiers, un certain nombre de règles doivent être respectées :

1. La première chose à laquelle il faut veiller est la présentation des données. Les variables doivent être en colonnes et les observations en lignes.
2. Les cases vides qui correspondent à des données manquantes doivent contenir les lettres `NA` en majuscule. Il est important de bien faire la distinction entre les vrais zéros (*i.e.* les grandeurs mesurées pour lesquelles un zéro a été obtenu), et les valeurs manquantes, c'est à dire pour lesquelles aucune valeur n'a pu être obtenue

(*e.g.* variable non mesurée pour un individu donné ou à une station donnée).
3. Il est généralement conseillé d'utiliser la première ligne du tableau pour stocker le nom des variables
4. Ne jamais utiliser de caractères spéciaux tels que #, $, %, ^, &, *, (,), {, }, [,], des accents, des cédilles des guillemets ou des apostrophes… Cela pourrait causer des erreurs lors de l'importation dans `R`. Si votre fichier en contient, faites une recherche (*via* le menu `Edition > Rechercher et remplacer...`) pour remplacer chaque instance par un caractère qui ne posera pas de problème.
5. Évitez les espaces dans vos noms de variables, d'observations ou de catégories et remplacez-les par des points ou des `_`.
6. Des noms courts pour les variables sont généralement plus faciles à manipuler par la suite.
7. La première valeur de votre tableau devrait toujours se trouver dans la cellule A1 du tableur. Autrement dit, il ne devrait jamais y avoir de lignes incomplètes ou de lignes de commentaires au-dessus des données, ou de colonne vide à gauche de votre tableau. D'ailleurs, il ne devrait jamais y avoir de commentaires à droite ou en dessous de vos données non plus.

4.2.2 Fichiers au format tableur (.xls ou .xlsx)

À titre d'exemple, téléchargez le fichier dauphin.xls et placez-le dans votre répertoire de travail. Ce jeu de données contient des résultats de dosages de différents métaux lourds (cadmium, cuivre et mercure) dans différents organes (foie et rein) de plusieurs dauphins communs *Delphinus delphis*. Les informations de taille, d'âge et de statut reproducteur sont également précisées. Ouvrez ce fichier dans un tableur. Vous constaterez que son format ne permet pas de l'importer tel quel dans `R` :

- Il contient des lignes vides inutiles au-dessus des données.
- Il contient des commentaires inutiles au-dessus des données.
- Les titres de colonnes sont complexes et contiennent des caractères spéciaux.

- Dans le tableau, les données manquantes sont représentées soit par des “`*`”, soit par des cellules vides.

Importer un tel jeu de données dans `R` par les méthodes classiques (c’est-à-dire sans utiliser RStudio et uniquement grâce aux fonctions de base de `R`) demanderait donc un gros travail de mise en forme préalable. Heureusement, `RStudio` et le package `readxl` facilitent grandement le processus.

Dans `RStudio`, localisez l’onglet `Files` situé dans le panneau en bas à droite de l’interface du logiciel. Dans ce panneau, cliquez sur le nom du fichier `Dauphin.xls`, puis, dans le menu qui s’affiche, choisissez `Import Dataset...` :

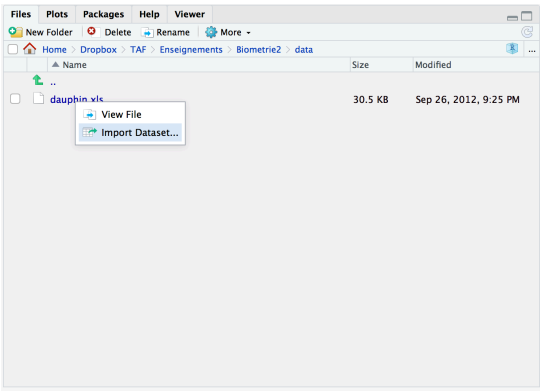


Figure 4.1: L’option `Import Dataset...` dans la fenêtre `Files` de RStudio

La nouvelle fenêtre qui s’ouvre est celle de l’“assistant d’importation” :

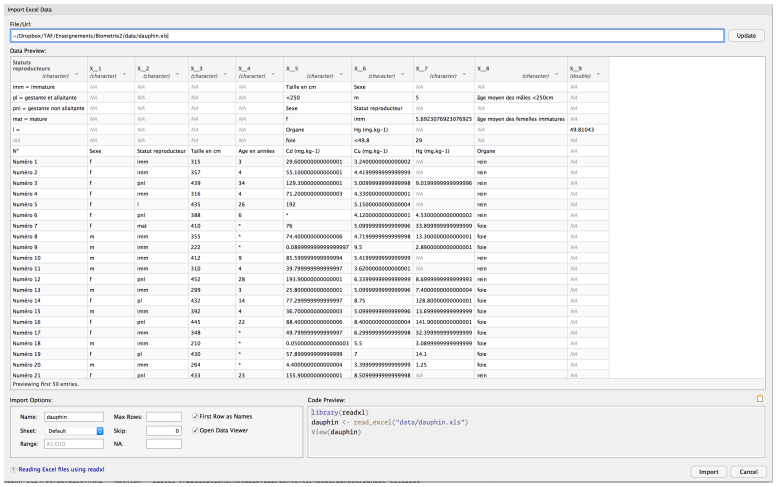


Figure 4.2: L'assistant d'importation de RStudio

Cette fenêtre contient plusieurs zones importantes :

1. `File/URL` (en haut) : lien vers le fichier contenant les données, sur votre ordinateur ou en ligne.
2. `Data Preview` : zone principale affichant les 50 premières lignes du fichier que l'on souhaite importer.
3. `Import Options` (en bas à gauche) : zone dans laquelle des options permettant d'importer les données correctement peuvent être spécifiées.
4. `Code Preview` (en bas à droite) : les lignes de codes que vous pourrez copier-coller dans votre script une fois les réglages corrects effectués.

Ici, nous constatons que les données ne sont pas au bon format. La première chose que nous pouvons faire est d'indiquer à `R` que nous souhaitons ignorer les 9 premières lignes du fichier. Ensuite, nous précisons à `RStudio` que l'étoile "`*`" a été utilisée pour indiquer des données manquantes :

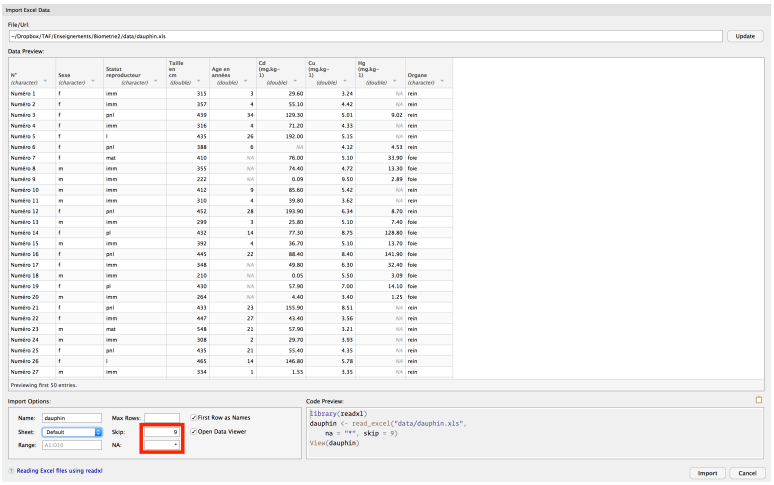


Figure 4.3: Les bons réglages pour ce fichier

Notez qu'à chaque fois que vous modifiez une valeur dans la zone `Import Options`, 2 choses se produisent simultanément :

1. La zone `Data Preview` est mise à jour. Cela permet de s'assurer que les changements effectués ont bien les effets escomptés.
2. La zone `Code Preview` est mise à jour. Cela permet de copier-coller dans votre script les commandes permettant d'importer correctement les données. Ici, voilà le code que nous devons ajouter à notre script :

```
dauphin <- read_excel("data/dauphin.xls", na = "*", skip = 9)
```

La commande `library(readxl)` est inutile puisque nous l'avons déjà saisie au début de ce chapitre. Nous disposons maintenant d'un nouvel objet nommé `dauphin`. Il est stocké sous la forme d'un `tibble` :

```
dauphin
```

```
# A tibble: 93 x 9
   `N°`       Sexe  `Statut reproducteur` `Taille en cm` `Age en années`
   <chr>      <chr> <chr>                          <dbl>            <dbl>
 1 Numéro 1   f     imm                              315                3
 2 Numéro 2   f     imm                              357                4
```

```
 3 Numéro 3  f     pnl                                439              34
 4 Numéro 4  f     imm                                316               4
 5 Numéro 5  f     l                                  435              26
 6 Numéro 6  f     pnl                                388               6
 7 Numéro 7  f     mat                                410              NA
 8 Numéro 8  m     imm                                355              NA
 9 Numéro 9  m     imm                                222              NA
10 Numéro 10 m     imm                                412               9
# i 83 more rows
# i 4 more variables: `Cd (mg.kg-1)` <dbl>, `Cu (mg.kg-1)` <dbl>,
#   `Hg (mg.kg-1)` <dbl>, Organe <chr>
```

Notez toutefois que les noms de colonnes complexes sont toujours présents. Avec de tels noms, les variables ne seront pas faciles à manipuler et les risques d'erreurs de frappes seront nombreux. Nous avons tout intérêt à les modifier à l'aide de la fonction `names()` :

```
names(dauphin) <- c("ID", "Sexe", "Statut", "Taille",
                    "Age", "Cd", "Cu", "Hg", "Organe")
dauphin
```

```
# A tibble: 93 x 9
   ID        Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <chr>     <chr> <chr>   <dbl> <dbl>  <dbl> <dbl> <dbl> <chr>
 1 Numéro 1  f     imm       315     3  29.6   3.24 NA    rein
 2 Numéro 2  f     imm       357     4  55.1   4.42 NA    rein
 3 Numéro 3  f     pnl       439    34 129.    5.01  9.02 rein
 4 Numéro 4  f     imm       316     4  71.2   4.33 NA    rein
 5 Numéro 5  f     l         435    26 192     5.15 NA    rein
 6 Numéro 6  f     pnl       388     6  NA     4.12  4.53 rein
 7 Numéro 7  f     mat       410    NA  76     5.1  33.9  foie
 8 Numéro 8  m     imm       355    NA  74.4   4.72 13.3  foie
 9 Numéro 9  m     imm       222    NA   0.09  9.5   2.89 foie
10 Numéro 10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

Enfin, vous pouvez également noter que certaines variables devraient être modifiées :

- Les variables `Sexe`, `Statut` (qui contient l'information de statut reproducteur des dauphins) et `Organe` (qui

indique dans quel organe les métaux ont été dosés) sont de type `<chr>`. L'idéal serait de disposer de facteurs puisqu'ils s'agit de variables catégorielles.
- La variable `ID` est totalement inutile puisqu'elle est parfaitement redondante avec le numéro de ligne. Nous pourrions donc la supprimer.
- Certaines catégories (ou niveaux) de la variable `Statut` devraient être ordonnées puisqu'elles reflètent une progression logique : `imm` (immature), `mat` (mature), `pnl` (pregnant non lactating), `pl` (pregnant lactating), `l` (lactating), `repos` (repos somatique).

Nous verrons dans les sections suivantes comment effectuer simplement ces différentes opérations.

4.2.3 Fichiers au format texte brut (.csv)

Nous allons utiliser les mêmes données que précédemment, mais cette fois-ci, elles sont contenues dans un fichier au format `.csv`. Téléchargez le fichier dauphin.csv (pour cela, faites un clic droit sur le lien et choisissez `Enregistrez la cible du lien sous...` ou `Télécharger le fichier lié sous...`, ou toute autre mention équivalente), placez-le dans votre répertoire de travail, et ouvrez-le avec le bloc notes Windows ou tout autre éditeur de texte brut disponible sur votre ordinateur. **Attention** : Microsoft Word n'est pas un éditeur de texte brut. Un fichier au format `.doc` ou `.docx` est illisible dans un éditeur de texte brut car outre le texte, ces formats de documents contiennent toutes les informations concernant la mise en forme du texte (polices de caractères, tailles, couleurs et autres attributs, présence de figures, de tableaux dans le document, etc.). Attention aussi à ne pas ouvrir vos fichiers `.csv` avec un tableur tel qu'Excel : la plupart du temps, Excel modifie sans le dire le format de ces fichiers (changement des symboles pour les décimales, ajout de caractères spéciaux et ou invisibles mal reconnus par `R`, etc.), ce qui cause toutes sortes de problèmes. Tenez vous-en bien à un éditeur de texte brut pour examiner le contenu des fichiers de ce type.

Les fichiers au format `.txt`, `.csv` et même `.R` (vos scripts !) sont des fichiers au format texte brut. Vous pouvez d'ailleurs essayer d'ouvrir `dauphin.csv` depuis `RStudio`, en allant dans

l'onglet `Files` (quart inférieur droit de l'interface de `RStudio`) puis en cliquant sur le nom du fichier et en choisissant `View File`. RStudio ouvre un nouvel onglet à côté de votre script vous permettant d'inspecter le contenu de ce fichier. Par rapport au fichier Excel, vous pouvez noter un certain nombre de différences :

1. Les colonnes sont séparées par des tabulations.
2. Les nombres décimaux utilisent la virgule (et non le point comme dans les pays anglo-saxons).
3. Les noms de colonnes ont déjà été corrigés/simplifiés par rapport au tableau d'origine.
4. Les valeurs manquantes sont toutes codées par des `NA`s.

Un travail d'édition du fichier `.xls` de départ a donc été réalisé en amont de l'enregistrement au format `.csv`.

Attention, à ce stade, vous avez ouvert un fichier au format texte brut dans `RStudio`, mais les données contenues dans ce fichier n'ont pas été importées pour autant. Pour les importer, on procède comme pour les fichiers au format tableur (voir section précédente).

On commence par cliquer sur `dauphin.csv` dans l'onglet `Files` de `RStudio`. On sélectionne ensuite `Import Dataset...` :

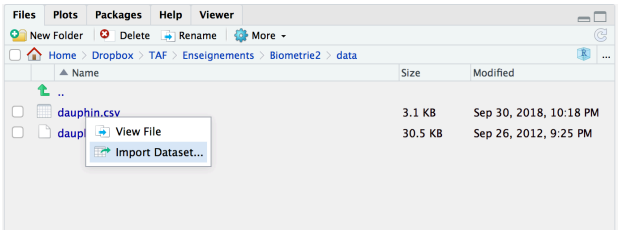


Figure 4.4: Importer un fichier `.csv` depuis l'onglet `Files` de RStudio

La fenêtre qui s'ouvre est en tous points identique à celle obtenue pour l'importation de fichiers tableurs :

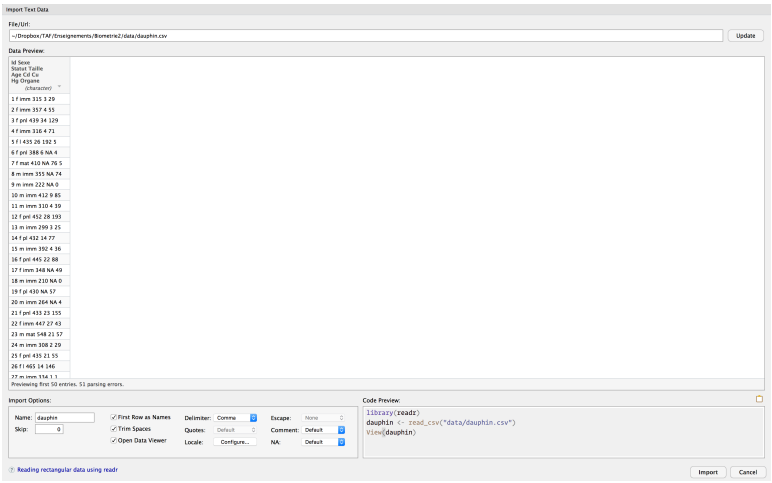


Figure 4.5: Importer un fichier `.csv` depuis l'onglet `Files` de RStudio

Nous voyons ici que par défaut, `RStudio` considère qu'une unique colonne est présente. En effet, les fichiers `.csv` utilisent généralement la virgule pour séparer les colonnes. Ce n'est pas le cas ici. Il nous faut donc sélectionner, dans le champ `Delimiter`, l'option `Tab` (tabulation) et non `Comma` (virgule).

À ce stade, chaque variable est maintenant reconnue comme telle, chaque variable occupe donc une colonne distincte. Mais les colonnes `Cd`, `Cu` et `Hg` ne contiennent pas les bonnes valeurs (vous pouvez le vérifier en consultant l'onglet `dauphin.csv` que vous avez ouvert un peu plus tôt à côté de votre script). La cause est simple : `R` s'attend à ce que les nombres décimaux utilisent le point en guise de symbole des décimales. Or, notre fichier `.csv` utilise la virgule. C'est une convention qui dépend du pays dans lequel vous vous trouvez, et de la langue de votre système d'exploitation (en langage technique, on parle de `Locale`). Le fichier `dauphin.csv` ayant été créé sur un ordinateur français, la virgule a été utilisée en guise de symbole des décimales. Pour l'indiquer au logiciel, cliquez sur `Locale > Configure...`, changez le `.` en `,` dans le champ `Decimal Mark` et validez en cliquant sur `Configure`.

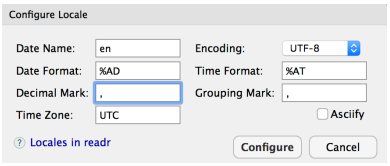


Figure 4.6: Changement du symbole utilisé pour les décimales

Les données sont maintenant au bon format, prêtes à être importées dans `RStudio`. Afin de ne pas écraser l'objet `dauphin` que nous avons créé à partir du fichier tableur un peu plus tôt, nous stockerons ces nouvelles données dans un objet nommé `dauphin2`. Pour cela, ajoutez un `2` au nom `dauphin` dans le champ `Name` en bas à gauche :

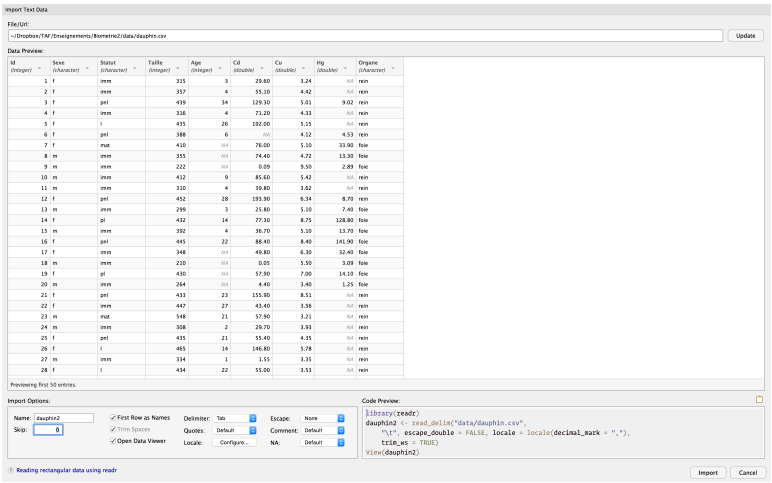


Figure 4.7: Les données, dans un format correct permettant l'importation

Nous n'avons plus qu'à copier-coller dans notre script le code généré automatiquement en bas à droite de la fenêtre (comme précédemment, la ligne `library(readr)` est inutile : nous avons déjà chargé ce package en début de chapitre).

```
dauphin2 <- read_delim("data/dauphin.csv",
    "\t", escape_double = FALSE, locale = locale(decimal_mark = ","),
    trim_ws = TRUE)
```

```
Rows: 93 Columns: 9
```

```
-- Column specification --------------------------------------------------------
Delimiter: "\t"
chr (3): Sexe, Statut, Organe
dbl (6): Id, Taille, Age, Cd, Cu, Hg

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Notez que :

1. C'est le package `readr` et non plus `readxl` qui est utilisé.
2. La fonction `read_delim()` a remplacé la fonction `read_excel()`. Il existe beaucoup d'autres fonctions selon le format de vos données (par exemple `read_csv()` et `read_csv2()`). Il est inutile de toutes les connaître dans la mesure où généralement, `RStudio` vous propose automatiquement la plus appropriée.
3. `R` indique de quelle façon les colonnes ont été "parsées", autrement dit, quelles fonctions ont été utilisées pour reconnaître le type des données présentes dans chaque colonne.

Toutes les fonctions permettant d'importer des données n'ont pas nécessairement le même comportement. Ainsi, si l'on compare les objets importés depuis le fichier tableur (`dauphin`) et depuis le fichier texte brut (`dauphin2`), le type de certaines variables peut être différent :

```
dauphin
```

```
# A tibble: 93 x 9
   ID        Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <chr>     <chr> <chr>   <dbl> <dbl>  <dbl> <dbl> <dbl> <chr>
 1 Numéro 1  f     imm       315     3  29.6   3.24 NA    rein
 2 Numéro 2  f     imm       357     4  55.1   4.42 NA    rein
 3 Numéro 3  f     pnl       439    34 129.    5.01  9.02 rein
 4 Numéro 4  f     imm       316     4  71.2   4.33 NA    rein
 5 Numéro 5  f     l         435    26 192     5.15 NA    rein
 6 Numéro 6  f     pnl       388     6  NA     4.12  4.53 rein
 7 Numéro 7  f     mat       410    NA  76     5.1  33.9  foie
 8 Numéro 8  m     imm       355    NA  74.4   4.72 13.3  foie
 9 Numéro 9  m     imm       222    NA   0.09  9.5   2.89 foie
```

```
10 Numéro 10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

```
dauphin2
```

```
# A tibble: 93 x 9
      Id Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <dbl> <chr> <chr>   <dbl> <dbl>  <dbl> <dbl> <dbl> <chr>
 1     1 f     imm       315     3  29.6   3.24 NA    rein
 2     2 f     imm       357     4  55.1   4.42 NA    rein
 3     3 f     pnl       439    34 129.    5.01  9.02 rein
 4     4 f     imm       316     4  71.2   4.33 NA    rein
 5     5 f     l         435    26 192     5.15 NA    rein
 6     6 f     pnl       388     6  NA     4.12  4.53 rein
 7     7 f     mat       410    NA  76     5.1  33.9  foie
 8     8 m     imm       355    NA  74.4   4.72 13.3  foie
 9     9 m     imm       222    NA   0.09  9.5   2.89 foie
10    10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

En particulier selon la version des packages que vous utilisez et les réglages spécifiques de vos systèmes d'exploitation, les variables `Taille` et `Age` sont parfois considérées comme réelles dans `dauphin` mais comme entières dans `dauphin2` (ce n'est pas le cas ici). Afin d'éviter les confusions dans la suite du document, nous allons supprimer `dauphin2` en tapant :

```
rm(dauphin2)
```

Taper `dauphin2` dans la console devrait maintenant produire une erreur :

```
dauphin2
```

```
Error in eval(expr, envir, enclos): objet 'dauphin2' introuvable
```

4.2.4 En cas de problème...

Il arrive parfois que l'importation de fichiers textes bruts par la méthode décrite ci-dessus échoue en raison d'un bug du

package `readr` qui gère mal la présence de caractères spéciaux (accents, cédilles, etc) dans le chemin des fichiers que l'on tente d'importer. À l'heure où j'écris ces lignes ce bug n'est toujours pas corrigé dans la version stable disponible au téléchargement sur les serveurs du CRAN. Il est donc utile de connaître une méthode alternative pour importer de tels fichiers dans `R`. Cette méthode repose sur "la mère de toutes les fonctions d'importation" : `read.table()`.

La fonction `read.table()` est à la base de la plupart des fonctions d'importation décrites dans ce chapitre. Il est donc important d'en connaître la syntaxe et les arguments les plus importants. Cette fonction requiert en général les arguments suivants :

1. Le chemin du fichier texte contenant les données à importer. Si le fichier se trouve dans votre répertoire de travail, il suffit de donner son nom. S'il est dans un sous-dossier de votre répertoire de travail, il faut donner le nom complet : `"sous_dossier/nom_du_fichier.csv"`.
2. `sep` : la spécification du symbole utilisé en guise de séparateur de colonnes dans le fichier texte. Cela peut-être la virgule (`sep = ","`), le point virgule (`sep = ";"`) ou encore la tabulation (`sep = "\t"`) selon les fichiers importés.
3. `dec` : la spécification du symbole utilisé en guise de symbole pour les décimales. Il n'est pas nécessaire de spécifier cet argument lorsque le symbole dans le fichier source est le point. Mais si c'est une virgule (comme c'est souvent le cas dans les pays francophones), il faut alors préciser `dec = ","`.
4. `header` : la première ligne du fichier source contient-elle des noms de variables. Si oui, il faut indiquer `header = TRUE`.

Ainsi, par exemple, pour le fichier `dauphin.csv` que j'ai placé dans un sous-dossier de mon répertoire de travail nommé `data`, on peut taper ceci :

```
dauph <- read.table("data/dauphin.csv",
                    sep = "\t",
                    dec = ",",
                    header = TRUE)
dauph
```

	Id	Sexe	Statut	Taille	Age	Cd	Cu	Hg	Organe
1	1	f	imm	315	3	29.60	3.24	NA	rein
2	2	f	imm	357	4	55.10	4.42	NA	rein
3	3	f	pnl	439	34	129.30	5.01	9.02	rein
4	4	f	imm	316	4	71.20	4.33	NA	rein
5	5	f	l	435	26	192.00	5.15	NA	rein
6	6	f	pnl	388	6	NA	4.12	4.53	rein
7	7	f	mat	410	NA	76.00	5.10	33.90	foie
8	8	m	imm	355	NA	74.40	4.72	13.30	foie
9	9	m	imm	222	NA	0.09	9.50	2.89	foie
10	10	m	imm	412	9	85.60	5.42	NA	rein
11	11	m	imm	310	4	39.80	3.62	NA	rein
12	12	f	pnl	452	28	193.90	6.34	8.70	rein
13	13	m	imm	299	3	25.80	5.10	7.40	foie
14	14	f	pl	432	14	77.30	8.75	128.80	foie
15	15	m	imm	392	4	36.70	5.10	13.70	foie
16	16	f	pnl	445	22	88.40	8.40	141.90	foie
17	17	f	imm	348	NA	49.80	6.30	32.40	foie
18	18	m	imm	210	NA	0.05	5.50	3.09	foie
19	19	f	pl	430	NA	57.90	7.00	14.10	foie
20	20	m	imm	264	NA	4.40	3.40	1.25	foie
21	21	f	pnl	433	23	155.90	8.51	NA	rein
22	22	f	imm	447	27	43.40	3.56	NA	rein
23	23	m	mat	548	21	57.90	3.21	NA	rein
24	24	m	imm	308	2	29.70	3.93	NA	rein
25	25	f	pnl	435	21	55.40	4.35	NA	rein
26	26	f	l	465	14	146.80	5.78	NA	rein
27	27	m	imm	334	1	1.55	3.35	NA	rein
28	28	f	l	434	22	55.00	3.53	NA	rein
29	29	f	pnl	387	6	90.10	4.17	NA	rein
30	30	f	repos	444	40	107.30	5.01	NA	rein
31	31	m	mat	581	18	164.30	5.69	NA	rein
32	32	f	imm	359	11	NA	3.97	NA	rein
33	33	f	imm	245	NA	0.07	5.80	1.30	foie
34	34	m	imm	346	4	34.40	2.65	21.60	foie
35	35	f	imm	370	NA	36.40	3.80	15.70	foie
36	36	f	l	432	27	80.10	3.96	NA	rein
37	37	m	imm	279	2	7.84	3.63	NA	rein
38	38	f	imm	316	3	34.20	3.21	NA	rein
39	39	m	imm	315	2	16.50	3.35	NA	rein
40	40	f	pnl	363	8	56.20	4.00	NA	rein
41	41	f	pnl	457	14	123.30	5.86	4.23	rein
42	42	m	imm	472	9	109.60	4.50	NA	rein

43	43	f	pl	442	16	193.10	5.25	6.38	rein
44	44	m	imm	422	10	75.10	6.90	53.15	foie
45	45	f	imm	193	1	NA	5.70	NA	rein
46	46	m	imm	324	4	31.60	4.50	7.90	foie
47	47	f	pnl	478	36	82.20	6.70	243.60	foie
48	48	f	pnl	451	NA	584.70	5.26	NA	rein
49	49	m	imm	245	NA	0.07	6.00	5.70	foie
50	50	f	pnl	405	NA	70.00	6.90	28.70	foie
51	51	m	imm	433	NA	55.20	7.50	52.50	foie
52	52	m	imm	326	NA	19.90	3.70	11.00	foie
53	53	f	pnl	440	22	84.00	9.10	207.40	foie
54	54	f	pnl	431	NA	90.30	4.64	NA	rein
55	55	f	pl	428	12	108.30	4.47	3.93	rein
56	56	f	imm	308	NA	50.80	5.00	27.00	foie
57	57	f	pnl	422	14	70.40	7.30	62.90	foie
58	58	f	pl	426	12	94.40	8.00	85.00	foie
59	59	f	imm	216	NA	0.01	4.20	3.53	foie
60	60	f	imm	314	NA	37.10	4.40	15.20	foie
61	61	f	pnl	394	9	78.50	6.20	55.20	foie
62	62	m	imm	400	5	89.10	4.83	NA	rein
63	63	f	pnl	416	NA	67.80	7.50	95.90	foie
64	64	f	imm	275	2	2.30	3.00	1.20	foie
65	65	m	imm	382	10	89.10	4.62	NA	rein
66	66	f	imm	320	6	NA	4.65	NA	rein
67	67	f	pl	418	12	89.10	4.26	5.44	rein
68	68	f	pnl	423	12	71.90	7.20	72.50	foie
69	69	f	l	407	11	64.40	5.10	39.00	foie
70	70	f	pl	459	22	76.20	10.70	178.20	foie
71	71	m	imm	215	NA	0.04	4.40	2.74	foie
72	72	m	imm	354	5	31.80	5.40	172.10	foie
73	73	m	imm	237	NA	0.07	7.10	1.29	foie
74	74	m	mat	513	18	44.30	5.80	49.80	foie
75	75	f	pl	431	15	80.80	9.40	145.20	foie
76	76	m	mat	519	15	71.30	4.41	NA	rein
77	77	f	pnl	386	8	918.80	7.38	NA	rein
78	78	f	imm	378	5	129.20	6.38	NA	rein
79	79	f	imm	342	4	89.40	4.68	NA	rein
80	80	m	mat	582	21	127.10	5.76	NA	rein
81	81	f	l	453	24	59.40	7.90	141.90	foie
82	82	f	imm	242	1	7.20	4.30	3.48	foie
83	83	f	imm	292	3	33.20	3.60	NA	rein
84	84	m	mat	545	25	89.30	2.67	NA	rein
85	85	f	imm	185	NA	0.01	5.50	2.58	foie

```
86 86    m    imm    340  NA  42.00  4.20   9.90   foie
87 87    m    imm    232   5   0.08  6.40   1.10   foie
88 88    m    mat    560  24  35.70  5.70  69.70   foie
89 89    m    imm    308  NA  38.20  4.40  11.50   foie
90 90    m    imm    227  NA   3.40  3.70   3.08   foie
91 91    f     pl    460  19  98.90  3.63     NA   rein
92 92    f      l    436  35  94.00  4.01     NA   rein
93 93    m    mat    582  27  82.20  3.39     NA   rein
```

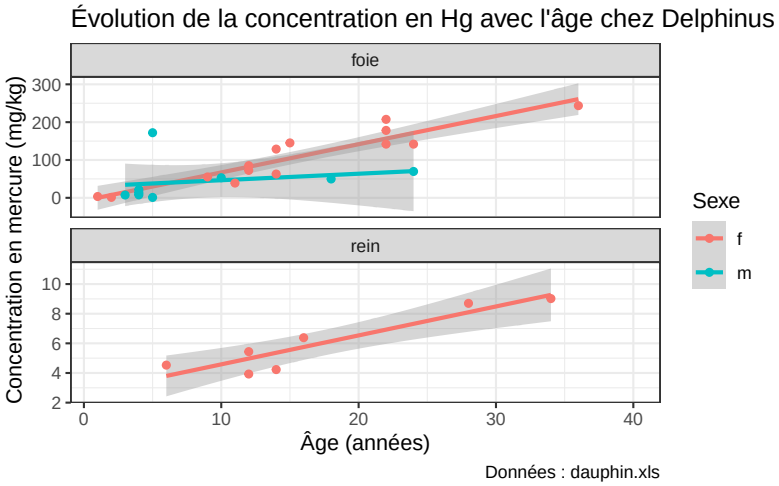
Puisque la fonction `read.table()` importe les données sous la forme d'un `data.frame`, il est nécessaire de transformer le tableau obtenu en `tibble` grâce à la fonction `as_tibble()` afin de bénéficier de tous les avantages de ce format d'objet.

```
dauph <- as_tibble(dauph)
dauph
```

```
# A tibble: 93 x 9
      Id Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <int> <chr> <chr>   <int> <int>  <dbl> <dbl> <dbl> <chr>
 1     1 f     imm       315     3  29.6   3.24 NA    rein
 2     2 f     imm       357     4  55.1   4.42 NA    rein
 3     3 f     pnl       439    34 129.    5.01  9.02 rein
 4     4 f     imm       316     4  71.2   4.33 NA    rein
 5     5 f     l         435    26 192     5.15 NA    rein
 6     6 f     pnl       388     6  NA     4.12  4.53 rein
 7     7 f     mat       410    NA  76     5.1  33.9  foie
 8     8 m     imm       355    NA  74.4   4.72 13.3  foie
 9     9 m     imm       222    NA   0.09  9.5   2.89 foie
10    10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

4.2.5 Exercices

Avec l'objet `dauphin`, produisez le graphique ci-dessous :



Rappel : les droites de régression avec leurs intervalles de confiance sont ajoutés grâce à la fonction `geom_smooth(method = "lm")`.

4.3 Le pipe |>

Avant d'entrer dans le vif du sujet, je souhaite introduire ici la notion de "pipe" (prononcer à l'anglo-saxonne). Le pipe est un opérateur que nous avons déjà vu apparaître à plusieurs reprises dans les chapitres précédents sans expliquer son fonctionnement.

Le pipe, noté `|>`, peut être obtenu en pressant les touches `ctrl + shift + M` (ou `cmd + shift + M` sous MacOS) de votre clavier. Il permet d'enchaîner logiquement des actions les unes à la suite des autres. Globalement, le pipe prend l'objet situé à sa gauche, et le transmet à la fonction situé à sa droite. En d'autres termes, les 2 expressions suivantes sont strictement équivalentes :

```
# Ici, "f" est une fonction quelconque, "x" et "y" sont 2 objets dont la fonction a besoi

# Il s'agit d'un exemple fictif : ne tapez pas ceci dans votre script !
f(x, y)
x |> f(y)
```

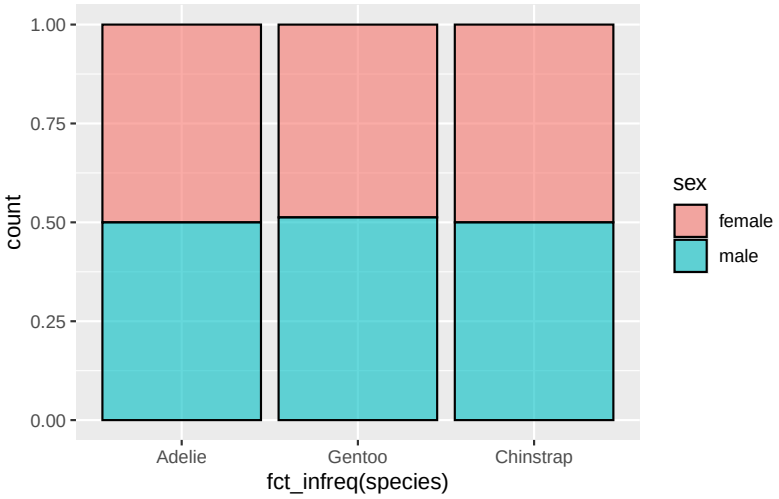
Travailler avec le pipe est très intéressant car toutes les fonctions de `dplyr` que nous allons décrire ensuite sont construites

autour de la même syntaxe : on leur fournit un `data.frame` (ou encore mieux, un `tibble`), elles effectuent une opération et renvoient un nouveau `data.frame` (ou un nouveau `tibble`). Il est ainsi possible de créer des groupes de commandes cohérentes qui permettent, grâce à l'enchaînement d'étapes simples, d'aboutir à des résultats complexes.

De la même façon que le `+` permet d'ajouter une couche supplémentaire à un graphique `ggplot2`, le pipe `|>` permet d'ajouter une opération supplémentaire dans un groupe de commandes.

Pour reprendre un exemple de la Section 3.8.1 sur les diagrammes bâtons empilés, nous avions utilisé ce code :

```
penguins |>
  filter(!is.na(sex)) |>
  ggplot(aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "fill")
```

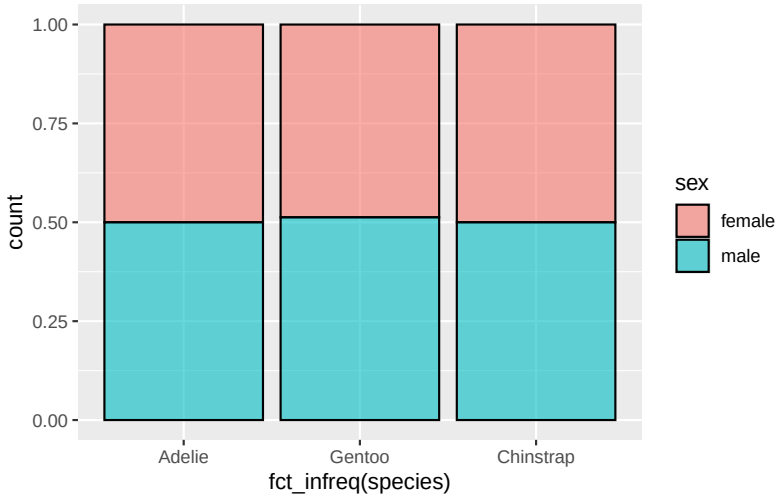


Ligne par ligne, voilà la signification de ce code :

- "Prend le tableau `penguins`, puis…"
- "transmets-le à la fonction `filter()` pour éliminer les lignes pour lequel le sexe est inconnu, puis…"
- "transmets le résultat à la fonction `ggplot()` pour en faire un graphique"

On aurait pu faire la même chose ainsi :

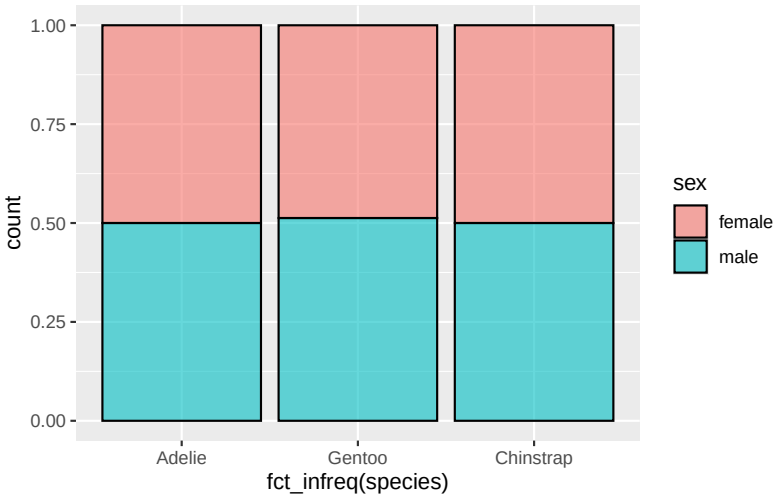
```r
penguins_clean <- filter(penguins, !is.na(sex))
ggplot(penguins_clean, aes(x = fct_infreq(species), fill = sex)) +
    geom_bar(alpha = 0.6, color = "black", position = "fill")
```



C'est strictement équivalent. La deuxième méthode à l'inconvénient de nous obliger à créer un objet intermédiaire (que j'ai ici nommé `penguins_clean`). Lorsque l'on a de nombreuses fonctions à enchaîner, il faut donc créer de nombreux objets intermédiaires dont nous n'avons besoin qu'une seule fois, ce qui peut être source de nombreuses erreurs.

Une troisième façon de procéder est la suivante :

```r
ggplot(filter(penguins, !is.na(sex)),
       aes(x = fct_infreq(species), fill = sex)) +
    geom_bar(alpha = 0.6, color = "black", position = "fill")
```



Cette fois, on ne crée plus d'objet intermédiaire, mais on intègre directement la fonction `filter()` à l'intérieur de la fonction `ggplot()`. Le code devient un peu moins lisible, et quand ça n'est pas deux fonctions mais 4, 5 ou plus que nous devons enchaîner, procéder ainsi est la garantie que des erreurs seront commises et qu'elles seront très difficiles à corriger.

On préfère donc toujours utiliser le pipe qui a le mérite de placer chaque fonction sur une nouvelle ligne, et de permettre une lecture plus simple du code, ligne par ligne, étape par étape, et non de façon imbriquée, de l'intérieur d'une commande vers l'extérieur :

```
penguins |>
  filter(!is.na(sex)) |>
  ggplot(aes(x = fct_infreq(species), fill = sex)) +
  geom_bar(alpha = 0.6, color = "black", position = "fill")
```

Notez bien qu'avec le pipe le premier argument de la fonction des fonctions `filter()` et `ggplot()` ont disparu : le pipe a fourni automatiquement à `filter()` les données du tableau `penguins`. Il a ensuite fourni automatiquement à `ggplot()` les données modifiées par la fonction `filter()`.

Comme pour le `+` de `ggplot2`, il est conseillé de placer un seul pipe par ligne, toujours à la fin, et de revenir à la ligne pour préciser l'étape suivante.

Toutes les commandes que nous utiliserons à partir de maintenant reposeront sur le pipe puisqu'il permet de rendre le

code plus lisible.

4.4 Les verbes du tripatouillage de données

Nous allons ici nous concentrer sur les fonctions les plus couramment utilisées pour manipuler et résumer des données. Nous verrons 4 verbes principaux, chacun correspondant à une fonction précise de `dplyr`. Chaque section de ce chapitre sera consacrée à la présentation d'un exemple utilisant un ou plusieurs de ces verbes.

Les 4 verbes sont :

1. `filter()` : choisir des lignes dans un tableau à partir de conditions spécifiques (filtrer).
2. `arrange()` : trie les lignes d'un tableau selon un ou plusieurs critères (arranger).
3. `select()` : sélectionner des colonnes d'un tableau.
4. `mutate()` : créer de nouvelles variables en transformant et combinant des variables existantes (muter).

Toutes ces fonctions, tous ces verbes, sont utilisés de la même façon : on prend un `data.frame`, grâce au pipe, on le transmet à l'une de ces fonctions dont on précise les arguments entre parenthèses, la fonction nous renvoie un nouveau tableau modifié. Évidemment, on peut enchaîner les actions pour modifier plusieurs fois le même tableau, c'est tout l'intérêt du pipe.

Enfin, gardez en tête qu'il existe beaucoup plus de fonctions dans `dplyr` que les 4 que nous allons détailler ici. Nous verrons parfois quelques variantes, mais globalement, maîtriser ces 4 fonctions simples devrait vous permettre d'aborder sereinement le premier semestre de la L2. Nous verrons d'autres fonctions de `dplyr` plus avancées, permettant notamment d'associer plusieurs `data.frame`s et de calculer des résumés numériques des données au semestre prochain, lorsque nous commencerons a nous intéresser à l'analyse statistique des données.

4.5 Filtrer des lignes avec `filter()`

4.5.1 Principe

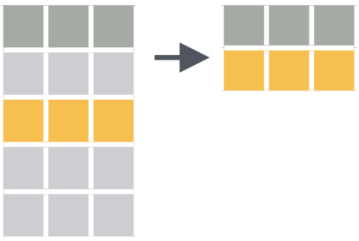


Figure 4.8: Schéma de la fonction `filter()` tiré de la 'cheat-sheet' de `dplyr` et `tidyr`

Comme son nom l'indique, `filter()` permet de filtrer des lignes en spécifiant un ou des critères de tri portant sur une ou plusieurs variables. Nous pouvons ainsi créer un nouveau tableau ne contenant que les données de l'espèce Adélie :

```
peng_adelie <- penguins |>
  filter(species == "Adelie")
```

La première ligne de code nous permet :

1. d'indiquer le nom du nouvel objet dans lequel les données modifiées seront stockées (ici, `peng_adelie`)
2. d'indiquer de quel objet les données doivent être extraites (`penguins`)
3. de passer cet objet à la fonction suivante avec un pipe `|>`

Le premier argument de la fonction `filter()` doit être le nom d'un `data.frame` ou d'un `tibble`. Ici, puisque nous utilisons le pipe, il est inutile de spécifier cet argument : c'est ce qui est placé à gauche du pipe qui est utilisé comme premier argument de la fonction `filter()`. Les arguments suivants constituent la ou les conditions qui doivent être respectées par les lignes du tableau de départ afin d'être intégrées au nouveau tableau de données.

4.5.2 Exercice

Créez un objet nommé `adelie_light` qui contiendra uniquement les données de l'espèce Adélie, et uniquement pour les individus pesant 3700 grammes ou moins. Indice : relisez la Section 1.3.4.2

Vérifiez que cet objet contient bien 81 lignes.

4.5.3 Les conditions logiques

Dans la Section 1.3.4.2, nous avons présenté en détail le fonctionnement des opérateurs de comparaison dans `R`. Relisez cette section si vous ne savez plus de quoi il s'agit. Les opérateurs de comparaison permettent de vérifier l'égalité ou l'inégalité entre des éléments. Ils renvoient `TRUE` ou `FALSE` et seront particulièrement utiles pour filtrer des lignes dans un tableau. Voici à nouveau la liste des opérateurs de comparaison usuels :

- `==` : égal à
- `!=` : différent de
- `>` : supérieur à
- `<` : inférieur à
- `>=` : supérieur ou égal à
- `<=` : inférieur ou égal à

À cette liste, nous pouvons ajouter quelques éléments utiles :

- `is.na()` : renvoie `TRUE` en cas de données manquantes.
- `!` : permet de tester le contraire d'une expression logique. Par exemple `!is.na()` renvoie `TRUE` s'il n'y a pas de données manquantes.
- `%in%` : permet de tester si l'élément de gauche est contenu dans la série d'éléments fournie à droite. Par exemple `2 %in% 1:5` renvoie `TRUE`, mais `2 %in% 5:10` renvoie `FALSE`.
- `|` : opérateur logique `OU`. Permet de tester qu'une condition `OU` une autre est remplie.
- `&` : opérateur logique `ET`. Permet de tester qu'une condition `ET` une autre sont remplies.

Voyons comment utiliser ces opérateurs avec la fonction `filter()`.

Dans le tableau `penguins`, quels sont les individus pour lesquels la masse n'a pas été mesurée ? Une bonne façon de le savoir est de regarder si, pour la variable `body_mass_g`, des données manquantes sont présentes :

```
penguins |>
  filter(is.na(body_mass_g))
```

```
# A tibble: 2 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen             NA            NA                NA          NA
2 Gentoo  Biscoe                 NA            NA                NA          NA
# i 2 more variables: sex <fct>, year <int>
```

Seules les lignes contenant `NA` dans la colonne `body_mass_g` sont retenues. Il y a donc 2 individus dont la masse est inconnue. D'ailleurs, pour ces individu, aucune mesure biométrique n'est disponible. il s'agit d'un manchot Adélie, et d'un manchot Gentoo, tous les deux de sexe inconnu.

Dans le même ordre d'idée, y a t-il des individus dont on ne connait pas le sexe mais dont on connait les mesures biométriques (au moins la masse) ? Là encore, une façon d'obtenir cette information est de sélectionner les individus dont le sexe est manquant, mais pour lesquels la masse n'est pas manquante :

```
penguins |>
  filter(is.na(sex),
         !is.na(body_mass_g))
```

```
# A tibble: 9 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           34.1          18.1               193        3475
2 Adelie  Torgersen           42            20.2               190        4250
3 Adelie  Torgersen           37.8          17.1               186        3300
4 Adelie  Torgersen           37.8          17.3               180        3700
```

```
5 Adelie  Dream               37.5          18.9               179        2975
6 Gentoo  Biscoe              44.5          14.3               216        4100
7 Gentoo  Biscoe              46.2          14.4               214        4650
8 Gentoo  Biscoe              47.3          13.8               216        4725
9 Gentoo  Biscoe              44.5          15.7               217        4875
# i 2 more variables: sex <fct>, year <int>
```

Notez l'utilisation du `!` pour la seconde condition. Nous récupérons ici les lignes pour lesquelles `body_mass_g` n'est pas `NA` et pour lesquelles `sex` est `NA`. Seules les lignes qui respectent cette double condition sont retenues. Cette syntaxe est équivalente à :

```
penguins |>
  filter(is.na(sex) & !is.na(body_mass_g))
```

```
# A tibble: 9 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           34.1          18.1               193        3475
2 Adelie  Torgersen           42            20.2               190        4250
3 Adelie  Torgersen           37.8          17.1               186        3300
4 Adelie  Torgersen           37.8          17.3               180        3700
5 Adelie  Dream               37.5          18.9               179        2975
6 Gentoo  Biscoe              44.5          14.3               216        4100
7 Gentoo  Biscoe              46.2          14.4               214        4650
8 Gentoo  Biscoe              47.3          13.8               216        4725
9 Gentoo  Biscoe              44.5          15.7               217        4875
# i 2 more variables: sex <fct>, year <int>
```

Dans la fonction `filter()`, séparer plusieurs conditions par des virgules signifie que seules les lignes qui remplissent toutes les conditions seront retenues. C'est donc l'équivalent du `ET` logique.

Enfin, pour illustrer l'utilisation de `|` (le `OU` logique) et de `%in%`, imaginons que nous souhaitions extraire les informations des individus de l'espèce Adélie qui vivent soit sur l'île Biscoe, soit sur l'île Dream, et dont le bec mesure moins de 42 mm de longueur :

```r
adel_small <- penguins |>
  filter(species == "Adelie",
         island == "Biscoe" | island == "Dream",
         bill_length_mm < 42)
adel_small
```

```
# A tibble: 91 x 8
   species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>           <dbl>         <dbl>             <int>       <int>
 1 Adelie  Biscoe           37.8          18.3               174        3400
 2 Adelie  Biscoe           37.7          18.7               180        3600
 3 Adelie  Biscoe           35.9          19.2               189        3800
 4 Adelie  Biscoe           38.2          18.1               185        3950
 5 Adelie  Biscoe           38.8          17.2               180        3800
 6 Adelie  Biscoe           35.3          18.9               187        3800
 7 Adelie  Biscoe           40.6          18.6               183        3550
 8 Adelie  Biscoe           40.5          17.9               187        3200
 9 Adelie  Biscoe           37.9          18.6               172        3150
10 Adelie  Biscoe           40.5          18.9               180        3950
# i 81 more rows
# i 2 more variables: sex <fct>, year <int>
```

Examinez ce tableau avec `View()` pour vérifier que la variable `island` contient bien uniquement les valeurs `Biscoe` et `Dream` correspondant aux 2 îles qui nous intéressent. Nous avons extrait ici les individus des îles Biscoe **et** Dream, pourtant, il nous a fallu utiliser le `OU` logique. Car chaque individu n'est issu que d'une unique île, or nous souhaitons récupérer toutes les lignes pour lesquelles l'île est soit `Biscoe`, soit `Dream` (l'une **ou** l'autre). Pour chaque ligne, les deux conditions ne peuvent pas être vraies l'une **et** l'autre en même temps. En revanche, on retient chaque ligne qui remplit la première condition **ou** la seconde.

Une autre solution pour obtenir le même tableau est de remplacer l'expression contenant `|` par une expression contenant `%in%` :

```r
adel_small2 <- penguins |>
  filter(species == "Adelie",
         island %in% c("Biscoe", "Dream"),
         bill_length_mm < 42)
```

```
adel_small2
```

```
# A tibble: 91 x 8
   species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>           <dbl>         <dbl>             <int>       <int>
 1 Adelie  Biscoe           37.8          18.3               174        3400
 2 Adelie  Biscoe           37.7          18.7               180        3600
 3 Adelie  Biscoe           35.9          19.2               189        3800
 4 Adelie  Biscoe           38.2          18.1               185        3950
 5 Adelie  Biscoe           38.8          17.2               180        3800
 6 Adelie  Biscoe           35.3          18.9               187        3800
 7 Adelie  Biscoe           40.6          18.6               183        3550
 8 Adelie  Biscoe           40.5          17.9               187        3200
 9 Adelie  Biscoe           37.9          18.6               172        3150
10 Adelie  Biscoe           40.5          18.9               180        3950
# i 81 more rows
# i 2 more variables: sex <fct>, year <int>
```

Ici, toutes les lignes du tableau dont la variable `island` est égale à un élément du vecteur `c("Biscoe", "Dream")` sont retenues. L'utilisation du `OU` logique peut être source d'erreur. Je préfère donc utiliser `%in%` qui me semble plus parlant. La fonction `identical()` nous confirme que les deux façons de faire produisent exactement le même résultat. Libre à vous de privilégier la méthode qui vous convient le mieux :

```
identical(adel_small, adel_small2)
```

```
[1] TRUE
```

4.6 Sélectionner des variables avec `select()`

Figure 4.9: Schéma de la fonction `select()` tiré de la 'cheat-sheet' de `dplyr` et `tidyr`

Il n'est pas rare de travailler avec des tableaux contenant des centaines, voir des milliers de colonnes. Dans de tels cas, il peut être utile de réduire le jeu de données aux variables qui vous intéressent. Le rôle de la fonction `select()` est de retenir uniquement les colonnes dont on a spécifié le nom, afin de recentrer l'analyse sur les variables utiles.

`select()` n'est pas particulièrement utile pour le jeu de données `penguins` puisqu'il ne contient que 8 variables. Toutefois, on peut malgré tout ces données pour comprendre le fonctionnement général de `select()`. Ainsi, pour sélectionner uniquement les colonnes `species`, `sex` et `body_mass_g`, on tape :

```
# Sélection de variables par leur nom
penguins |>
  select(species, sex, body_mass_g)
```

```
# A tibble: 344 x 3
   species sex    body_mass_g
   <fct>   <fct>        <int>
 1 Adelie  male          3750
 2 Adelie  female        3800
 3 Adelie  female        3250
 4 Adelie  <NA>            NA
 5 Adelie  female        3450
 6 Adelie  male          3650
```

```
 7 Adelie  female        3625
 8 Adelie  male          4675
 9 Adelie  <NA>          3475
10 Adelie  <NA>          4250
# i 334 more rows
```

Pour retenir des colonnes qui sont côte à côte dans le tableau de départ, on peut utiliser l'opérateur `:` pour les sélectionner :

```
# Sélection de toutes les variables entre `island` et `bill_depth_mm` (inclues)
penguins |>
  select(island:bill_depth_mm)
```

```
# A tibble: 344 x 3
   island    bill_length_mm bill_depth_mm
   <fct>              <dbl>         <dbl>
 1 Torgersen           39.1          18.7
 2 Torgersen           39.5          17.4
 3 Torgersen           40.3          18
 4 Torgersen           NA            NA
 5 Torgersen           36.7          19.3
 6 Torgersen           39.3          20.6
 7 Torgersen           38.9          17.8
 8 Torgersen           39.2          19.6
 9 Torgersen           34.1          18.1
10 Torgersen           42            20.2
# i 334 more rows
```

À l'inverse, si on veut supprimer certaines colonnes, on peut utiliser la notation `-` :

```
# Sélection de toutes les variables de `penguins` à l'exception
# de celles comprises entre `island` et `bill_depth_mm` (inclues)
penguins |>
  select(-(island:bill_depth_mm))
```

```
# A tibble: 344 x 5
   species flipper_length_mm body_mass_g sex     year
   <fct>               <int>       <int> <fct>  <int>
 1 Adelie                181        3750 male    2007
```

```
 2 Adelie                186        3800 female  2007
 3 Adelie                195        3250 female  2007
 4 Adelie                 NA          NA <NA>    2007
 5 Adelie                193        3450 female  2007
 6 Adelie                190        3650 male    2007
 7 Adelie                181        3625 female  2007
 8 Adelie                195        4675 male    2007
 9 Adelie                193        3475 <NA>    2007
10 Adelie                190        4250 <NA>    2007
# i 334 more rows
```

Il y a beaucoup de fonctions permettant de sélectionner des variables dont les noms respectent certains critères. Par exemple :

- `starts_with("abc")` : renvoie toutes les variables dont les noms commencent par “abc”
- `ends_with("xyz")` : renvoie toutes les variables dont les noms se terminent par “xyz”
- `contains("ijk")` : renvoie toutes les variables dont les noms contiennent “ijk”

Il en existe beaucoup d’autres. Vous pouvez consulter l’aide de `?select()` pour en savoir plus.

Ainsi, il est par exemple possible d’extraire toutes les variables contenant le mot “mm” ainsi :

```
penguins |>
  select(contains("mm"))
```

```
# A tibble: 344 x 3
   bill_length_mm bill_depth_mm flipper_length_mm
            <dbl>         <dbl>             <int>
 1           39.1          18.7               181
 2           39.5          17.4               186
 3           40.3          18                 195
 4           NA            NA                  NA
 5           36.7          19.3               193
 6           39.3          20.6               190
 7           38.9          17.8               181
 8           39.2          19.6               195
 9           34.1          18.1               193
```

```
10           42            20.2                190
# i 334 more rows
```

Évidemment, le tableau `penguins` n'est pas modifié par cette opération : il contient toujours les 8 variables de départ. Pour travailler avec ces tableaux de données contenant moins de variables, il faut les stocker dans un nouvel objet en leur donnant un nom :

```
measures <- penguins |>
  select(contains("mm"))
```

Enfin, on peut utiliser `select()` pour renommer des variables. Mais ce n'est que rarement utile car `select()` élimine toutes les variables qui n'ont pas été explicitement nommées :

```
penguins |>
  select(species:island,
         b_length = bill_length_mm,
         flipper = flipper_length_mm)
```

```
# A tibble: 344 x 4
   species island    b_length flipper
   <fct>   <fct>        <dbl>   <int>
 1 Adelie  Torgersen     39.1     181
 2 Adelie  Torgersen     39.5     186
 3 Adelie  Torgersen     40.3     195
 4 Adelie  Torgersen     NA        NA
 5 Adelie  Torgersen     36.7     193
 6 Adelie  Torgersen     39.3     190
 7 Adelie  Torgersen     38.9     181
 8 Adelie  Torgersen     39.2     195
 9 Adelie  Torgersen     34.1     193
10 Adelie  Torgersen     42       190
# i 334 more rows
```

Il est donc généralement préférable d'utiliser `rename()` pour renommer certaines variables sans en éliminer aucune :

```
penguins |>
  rename(b_length = bill_length_mm,
         flipper = flipper_length_mm)
```

```
# A tibble: 344 x 8
   species island    b_length bill_depth_mm flipper body_mass_g sex     year
   <fct>   <fct>        <dbl>         <dbl>   <int>       <int> <fct>  <int>
 1 Adelie  Torgersen     39.1          18.7     181        3750 male    2007
 2 Adelie  Torgersen     39.5          17.4     186        3800 female  2007
 3 Adelie  Torgersen     40.3          18       195        3250 female  2007
 4 Adelie  Torgersen     NA            NA        NA          NA <NA>    2007
 5 Adelie  Torgersen     36.7          19.3     193        3450 female  2007
 6 Adelie  Torgersen     39.3          20.6     190        3650 male    2007
 7 Adelie  Torgersen     38.9          17.8     181        3625 female  2007
 8 Adelie  Torgersen     39.2          19.6     195        4675 male    2007
 9 Adelie  Torgersen     34.1          18.1     193        3475 <NA>    2007
10 Adelie  Torgersen     42            20.2     190        4250 <NA>    2007
# i 334 more rows
```

4.7 Créer de nouvelles variables avec `mutate()`

4.7.1 Principe

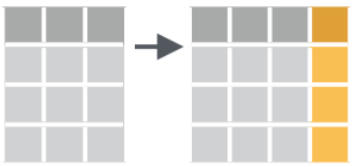


Figure 4.10: Schéma de la fonction `mutate()` tiré de la 'cheat-sheet' de `dplyr` et `tidyr`

La fonction `mutate()` permet de créer de nouvelles variables à partir des variables existantes, ou de modifier des variables déjà présentes dans un jeu de données. Il est en effet fréquent d'avoir besoin de calculer de nouvelles variables, souvent plus informatives que les variables disponibles.

Voyons un exemple. À partir de `penguins`, nous allons calculer 1 nouvelles variable et en modifier une autre :

1. `ratio` : le rapport entre la longueur du bec et son épaisseur. Cela nous donnera un indice de la compacité du

bec. Des valeurs faibles de ce ratio un bec très trapu, alors que des valeurs fortes indiqueront un bec très effilé
2. `mass_kg` : la masse, qui est ici exprimée en grammes sera transformée en kilogrammes par une simple division par 1000

```
penguins |>
  mutate(ratio = bill_length_mm / bill_depth_mm,
         mass_kg = body_mass_g / 1000)
```

```
# A tibble: 344 x 10
   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
 1 Adelie  Torgersen           39.1          18.7               181        3750
 2 Adelie  Torgersen           39.5          17.4               186        3800
 3 Adelie  Torgersen           40.3          18                 195        3250
 4 Adelie  Torgersen           NA            NA                  NA          NA
 5 Adelie  Torgersen           36.7          19.3               193        3450
 6 Adelie  Torgersen           39.3          20.6               190        3650
 7 Adelie  Torgersen           38.9          17.8               181        3625
 8 Adelie  Torgersen           39.2          19.6               195        4675
 9 Adelie  Torgersen           34.1          18.1               193        3475
10 Adelie  Torgersen           42            20.2               190        4250
# i 334 more rows
# i 4 more variables: sex <fct>, year <int>, ratio <dbl>, mass_kg <dbl>
```

Si on souhaite conserver uniquement les variables nouvellement créées par `mutate()`, on peut utiliser `transmute()` :

```
penguins |>
  transmute(ratio = bill_length_mm / bill_depth_mm,
            mass_kg = body_mass_g / 1000)
```

```
# A tibble: 344 x 2
   ratio mass_kg
   <dbl>   <dbl>
 1  2.09    3.75
 2  2.27    3.8
 3  2.24    3.25
 4 NA      NA
 5  1.90    3.45
```

```
 6  1.91    3.65
 7  2.19    3.62
 8  2       4.68
 9  1.88    3.48
10  2.08    4.25
# i 334 more rows
```

Et comme toujours, pour pouvoir réutiliser ces données, on leur donne un nom :

```
pengu_ratio <-  penguins |>
  transmute(ratio = bill_length_mm / bill_depth_mm,
            mass_kg = body_mass_g / 1000)
```

4.7.2 Transformer des variables en facteurs

Une autre opération fréquente possible grâce à la fonction `mutate()` est la transformation d'une ou plusieurs variables d'un tableau en facteur avec la fonction `factor()`. Plusieurs variables du tableau `dauphin`, importé plus tôt, devrait être transformées en facteur :

```
dauphin
```

```
# A tibble: 93 x 9
   ID        Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <chr>     <chr> <chr>   <dbl> <dbl>  <dbl> <dbl> <dbl> <chr>
 1 Numéro 1  f     imm       315     3  29.6   3.24 NA    rein
 2 Numéro 2  f     imm       357     4  55.1   4.42 NA    rein
 3 Numéro 3  f     pnl       439    34 129.    5.01  9.02 rein
 4 Numéro 4  f     imm       316     4  71.2   4.33 NA    rein
 5 Numéro 5  f     l         435    26 192     5.15 NA    rein
 6 Numéro 6  f     pnl       388     6  NA     4.12  4.53 rein
 7 Numéro 7  f     mat       410    NA  76     5.1  33.9  foie
 8 Numéro 8  m     imm       355    NA  74.4   4.72 13.3  foie
 9 Numéro 9  m     imm       222    NA   0.09  9.5   2.89 foie
10 Numéro 10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

C'est le cas des variables `Sexe`, `Statut` et `Organe`. Par ailleurs, la variable `ID` pourrait être supprimée puisqu'elle n'apporte

aucune information est est parfaitement redondante avec les numéros de ligne du tableau. Voyons comment réaliser toutes ces actions :

```
dauphin_clean <- dauphin |>
  select(-ID) |>                  # Suppression de la colonne ID, puis
  mutate(Sexe = factor(Sexe),      # Transformation de Sexe en facteur
         Organe = factor(Organe), # Transformation d'Organe en facteur
         Statut = factor(Statut,  # Transformation de Statut en facteur
                         levels = c("imm", "mat", "pnl", "pl", "l", "repos")))
```

L'objet `dauphin_clean` contient les résultats de nos manipulations :

```
dauphin_clean
```

```
# A tibble: 93 x 8
   Sexe  Statut Taille   Age     Cd    Cu    Hg Organe
   <fct> <fct>   <dbl> <dbl>  <dbl> <dbl> <dbl> <fct>
 1 f     imm       315     3  29.6   3.24 NA    rein
 2 f     imm       357     4  55.1   4.42 NA    rein
 3 f     pnl       439    34 129.    5.01  9.02 rein
 4 f     imm       316     4  71.2   4.33 NA    rein
 5 f     l         435    26 192     5.15 NA    rein
 6 f     pnl       388     6  NA     4.12  4.53 rein
 7 f     mat       410    NA  76     5.1  33.9  foie
 8 m     imm       355    NA  74.4   4.72 13.3  foie
 9 m     imm       222    NA   0.09  9.5   2.89 foie
10 m     imm       412     9  85.6   5.42 NA    rein
# i 83 more rows
```

Vous notez que `ID` a disparu et que les 3 variables modifiées sont maintenant bel et bien des facteurs. Vous avez probablement remarqué également que pour la variable `Statut`, la syntaxe que j'ai utilisée est légèrement différente de celle des variables `Sexe` et `Organe`. Pour en comprendre la raison, tapez ceci pour afficher le contenu de ces facteurs :

```
dauphin_clean$Sexe
```

```
 [1] f f f f f f f m m m m f m f m f f m f m f f m m f f m f f f m f f m f f m f
[39] m f f m f m f m f f m f m m f f f f f f f f f m f f m f f f f f m m m m f m
```

```
[77] f f f m f f f m f m m m m m f f m
Levels: f m
```

```
dauphin_clean$Organe
```

```
 [1] rein rein rein rein rein rein foie foie foie rein rein rein foie foie foie
[16] foie foie foie foie foie rein rein rein rein rein rein rein rein rein rein
[31] rein rein foie foie foie rein rein rein rein rein rein rein rein rein foie rein
[46] foie foie rein foie foie foie foie foie rein rein foie foie foie foie foie
[61] foie rein foie foie rein rein rein foie foie foie foie foie foie foie foie
[76] rein rein rein rein rein foie foie rein rein foie foie foie foie foie foie
[91] rein rein rein
Levels: foie rein
```

```
dauphin_clean$Statut
```

```
 [1] imm   imm   pnl   imm   l     pnl   mat   imm   imm   imm   imm   pnl
[13] imm   pl    imm   pnl   imm   imm   pl    imm   pnl   imm   mat   imm
[25] pnl   l     imm   l     pnl   repos mat   imm   imm   imm   imm   l
[37] imm   imm   imm   pnl   pnl   imm   pl    imm   imm   imm   pnl   pnl
[49] imm   pnl   imm   imm   pnl   pnl   pl    imm   pnl   pl    imm   imm
[61] pnl   imm   pnl   imm   imm   imm   pl    pnl   l     pl    imm   imm
[73] imm   mat   pl    mat   pnl   imm   imm   mat   l     imm   imm   mat
[85] imm   imm   imm   mat   imm   imm   pl    l     mat
Levels: imm mat pnl pl l repos
```

Pour les 2 premiers facteurs, les niveaux des facteurs (ou modalités) sont classés par ordre alphabétique. Ainsi, pour le facteur `Sexe`, la catégorie `f` (femelle) apparaît avant `m` (mâles) dans la liste des niveaux (`Levels: ...`). Pour le facteur `Organe`, la modalité `foie` apparaît avant la modalité `rein`. L'ordre des modalités d'un facteur est celui qui sera utilisé par défaut pour ordonner les catégories sur les axes d'un graphique ou dans les légendes. L'ordre alphabétique convient parfaitement pour le `Sexe` ou l'`Organe` puisqu'il n'y a pas, pour ces facteurs, d'ordre dans les modalités.

Pour le facteur `Statut` en revanche, l'ordre importe, car il reflète des stades qui se succèdent logiquement au cours de la vie des individus (et des femelles plus particulièrement).

Sur un graphique, on souhaite donc que ces catégories apparaissent dans un ordre bien précis, différent de l'ordre alphabétique. C'est la raison pour laquelle, lorsque l'on crée un facteur avec la fonction `factor()`, on peut spécifier explicitement un ordre pour les catégories grâce à l'argument `levels =`. Il suffit ensuite de fournir un vecteur contenant le nom de chaque catégorie, dans l'ordre souhaité.

4.7.3 Exercices

1. Dans `ggplot2` le jeu de données `mpg` contient des informations sur 234 modèles de voitures. Examinez ce jeu de données avec la fonction `View()` et consultez son fichier d'aide pour savoir à quoi correspondent les différentes variables. Quelle(s) variable(s) nous renseignent sur la consommation des véhicules ? À quoi correspond la variable `disp` ?
2. La consommation est donnée en miles par gallon. Créez une nouvelle variable `conso` qui contiendra la consommation sur autoroute, exprimée en nombre de litres pour 100 kilomètres.
3. Faîtes un graphique présentant la relation entre la cylindrée en litres et la consommation sur autoroute exprimée en nombre de litres pour 100 kilomètres. Vous exclurez les véhicules dont la `classe` est `2seater` de ce graphique (il s'agit de voitures de sports très compactes qu'il est difficile de mesurer aux autres). Sur votre graphique, la couleur devrait représenter le type de véhicule. Vous ajouterez une droite de régression en utilisant `geom_smooth(method = "lm")`. Votre graphique devrait ressembler à ceci :

```
`geom_smooth()` using formula = 'y ~ x'
```

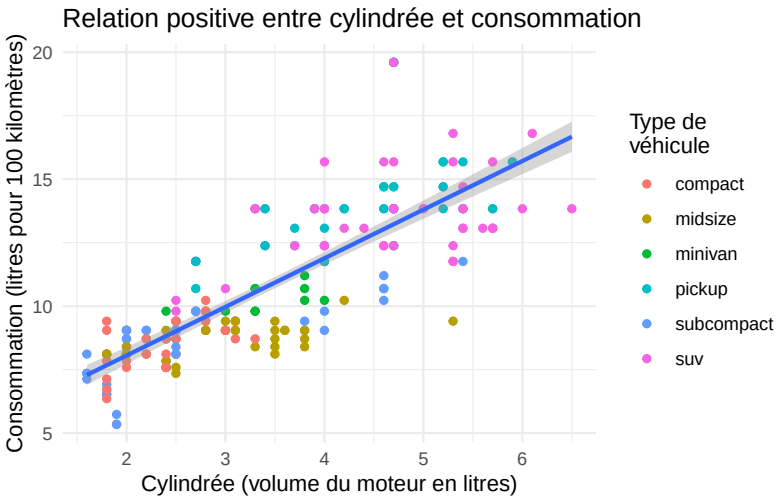


Figure 4.11: Consommation en fonction de la cylindrée

4. Ce graphique présente-t-il correctement l'ensemble des données de ces 2 variables ? Pourquoi ? Comparez le graphique de la question 3 ci-dessus et le graphique présenté ci-dessous. Selon vous, quels arguments et/ou fonctions ont été modifiés pour arriver à ce nouveau graphique ? Quels sont les avantages et les inconvénients de ce graphique par rapport au précédent ?

```
`geom_smooth()` using formula = 'y ~ x'
```

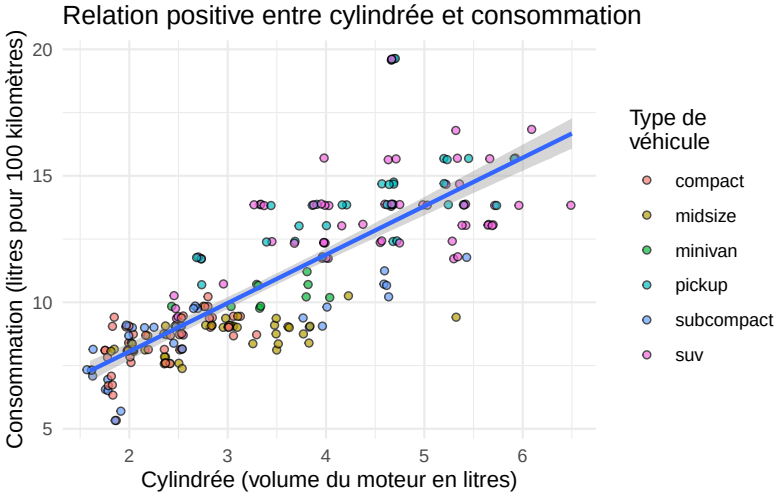


Figure 4.12: Consommation en fonction de la cylindrée

4.8 Trier des lignes avec `arrange()`

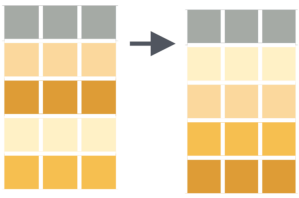


Figure 4.13: Schéma de la fonction `arrange()` tiré de la 'cheatsheet' de `dplyr` et `tidyr`

La fonction `arrange()` permet de trier des tableaux en ordonnant les éléments d'une ou plusieurs colonnes. Les tris peuvent être en ordre croissants (c'est le cas par défaut) ou décroissants (grâce à la fonction `desc()`, abréviation de "descending").

`arrange()` fonctionne donc comme `filter()`, mais au lieu de sélectionner des lignes, cette fonction change leur ordre. Il faut lui fournir le nom d'un tableau et au minimum le nom d'une variable selon laquelle le tri doit être réalisé. Si plusieurs variables sont fournies, chaque variable supplémentaire permet de résoudre les égalités. Ainsi, pour ordonner le tableau `penguins` par ordre croissant d'épaisseur de bec (`bill_depth_mm`), on tape :

```
penguins |>
  arrange(bill_depth_mm)
```

```
# A tibble: 344 x 8
   species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>           <dbl>         <dbl>             <int>       <int>
 1 Gentoo  Biscoe           42.9          13.1               215        5000
 2 Gentoo  Biscoe           46.1          13.2               211        4500
 3 Gentoo  Biscoe           44.9          13.3               213        5100
 4 Gentoo  Biscoe           43.3          13.4               209        4400
 5 Gentoo  Biscoe           46.5          13.5               210        4550
 6 Gentoo  Biscoe           42            13.5               210        4150
 7 Gentoo  Biscoe           44            13.6               208        4350
 8 Gentoo  Biscoe           40.9          13.7               214        4650
 9 Gentoo  Biscoe           45.5          13.7               214        4650
```

```
10 Gentoo  Biscoe           42.6          13.7               213        4950
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Notez que la variable `dbill_depth_mm` est maintenant triée en ordre croissant. Notez également que 2 individus ont un bec dont l'épaisseur vaut exactement 13,5 mm. Comparez le tableau précédent avec celui-ci :

```
penguins |>
  arrange(bill_depth_mm, bill_length_mm)
```

```
# A tibble: 344 x 8
   species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>   <fct>           <dbl>         <dbl>             <int>       <int>
 1 Gentoo  Biscoe           42.9          13.1               215        5000
 2 Gentoo  Biscoe           46.1          13.2               211        4500
 3 Gentoo  Biscoe           44.9          13.3               213        5100
 4 Gentoo  Biscoe           43.3          13.4               209        4400
 5 Gentoo  Biscoe           42            13.5               210        4150
 6 Gentoo  Biscoe           46.5          13.5               210        4550
 7 Gentoo  Biscoe           44            13.6               208        4350
 8 Gentoo  Biscoe           40.9          13.7               214        4650
 9 Gentoo  Biscoe           42.6          13.7               213        4950
10 Gentoo  Biscoe           42.7          13.7               208        3950
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Les lignes des 2 individus dont l'épaisseur du bec vaut 13,5 mm ont été inversées : la variable `bill_length_mm` a été utilisée pour ordonner les lignes en cas d'égalité de la variable `bill_depth_mm`.

Comme indiqué plus haut, il est possible de trier les données par ordre décroissant :

```
penguins |>
  arrange(desc(bill_depth_mm))
```

```
# A tibble: 344 x 8
   species   island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
   <fct>     <fct>             <dbl>         <dbl>             <int>       <int>
```

```
 1 Adelie    Torgers~           46           21.5                194        4200
 2 Adelie    Torgers~           38.6         21.2                191        3800
 3 Adelie    Dream              42.3         21.2                191        4150
 4 Adelie    Torgers~           34.6         21.1                198        4400
 5 Adelie    Dream              39.2         21.1                196        4150
 6 Adelie    Biscoe             41.3         21.1                195        4400
 7 Chinstrap Dream              54.2         20.8                201        4300
 8 Adelie    Torgers~           42.5         20.7                197        4500
 9 Adelie    Biscoe             39.6         20.7                191        3900
10 Chinstrap Dream              52           20.7                210        4800
# i 334 more rows
# i 2 more variables: sex <fct>, year <int>
```

Cela est particulièrement utile après l'obtention de résumés groupés (obtenus avec la fonction `count()`) pour connaître la catégorie la plus représentée. Par exemple, si nous souhaitons connaître l'espèce et le sexe les plus fréquemment observés, on peut procéder ainsi :

1. prendre le tableau `penguins`, *puis,*
2. compter le nombre d'observation par espèce et sexe avec la fonction `count`, *puis,*
3. trier les données par effectif décroissant.

```
penguins |>
  count(species, sex) |>
  arrange(desc(n))
```

```
# A tibble: 8 x 3
  species   sex        n
  <fct>     <fct>  <int>
1 Adelie    female    73
2 Adelie    male      73
3 Gentoo    male      61
4 Gentoo    female    58
5 Chinstrap female    34
6 Chinstrap male      34
7 Adelie    <NA>       6
8 Gentoo    <NA>       5
```

Deux catégories sont aussi fréquemment observées l'une que l'autre : les mâles et femelles de l'espèce Adélie, pour lesquels 73 individus ont été observés.

Références

Gorman, Kristen B., Tony D. Williams, et William R Fraser. 2014. « Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis) ». *PLOS ONE* 9 (mars): 1-14. https://doi.org/10.1371/journal.pone.0090081.

Horst, Allison, Alison Hill, et Kristen Gorman. 2022. *palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. https://allisonhorst.github.io/palmerpenguins/.

Jeppson, Haley, Heike Hofmann, et Di Cook. 2021. *ggmosaic: Mosaic Plots in the ggplot2 Framework*. https://github.com/haleyjeppson/ggmosaic.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, Dewey Dunnington, et Teun van den Brand. 2024. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. https://ggplot2.tidyverse.org.

Wilkinson, Leland. 2005. *The grammar of graphics*. 2nd éd. New-York: Springer-Verlag. https://www.springer.com/us/book/9780387245447.