# COMP151: Programming for Scientists (in Python)

Jeffrey Elkner Allen B. Downey Chris Meyers Brendan McCane
Nick Meek Iain Hewson

February 21, 2024

# Contents

# License

# Course Information

Welcome to COMP151. In this paper you will learn that programming is not only a fun activity but an essential skill for scientific research. Python is a great programming language to learn because it's accessible, versatile, and popular with scientists. In today's society, computer programs play a central role in almost all scientific work. Therefore, it is important for scientists to have coding skills. We hope this class helps you become a proficient programmer and empower you on your journey towards mastery.

If this is your first time at university, welcome! University is different from school in many ways, one of which is that you have a lot more freedom to structure your time. The most important thing for success is showing up and doing the work, so if you can do that, you'll do great!

In COMP151 we expect students to:

- Attend all lectures and read the coursebook.
- Attend all your scheduled lab sessions.
- Attempt one mastery level per week.
- Sit the practical tests as scheduled.
- If you are struggling ask for help.
- If you can't attend class let us know as soon as possible.

We hope you enjoy COMP151

The COMP151 Teaching Team,

Brendon and Karen.

## Course Structure

In COMP151, we use a mastery-based approach to teaching programming. This means that you will learn and practice skills and techniques in a specific order, with each new topic building upon your understanding of previous ones. As you progress through the course, **it's important to complete each topic before moving on to the next one**. While the lectures will follow a fixed schedule, your lab work may move at a different pace. If you find yourself getting ahead of the lectures, that's great! **Just make sure not to fall too far behind, as it can be difficult to catch up**. Since the course is self-paced, different students may reach different parts of the material by the end of the semester. We expect all students to complete up to and including Lesson 13, but you should aim to go as far as you can.

## Assessment

There are a total of 8 Mastery Levels and 2 Practical Tests in this course. The Mastery Levels are worth 60% of your final grade, and each Practical Test is worth 20%.

The Mastery Levels are pass/fail. In order to attempt the next level, you must successfully complete the previous one (except Levels 7 and 8 which may be completed in in either order).

You can retry a Mastery Level as many times as you like, but only once per day. If you pass a Mastery Level, you get all the marks available for that level. If you fail a Mastery Level, you don't get any marks but you may resit the following day.

Each of the Mastery Levels are summarised below in Figure 1.

| Mastery Levels | | Content | | Grade Progression | | | |
|---|---|---|---|---|---|---|---|
| | | Lesson(s) | Questions | Marks | A-Track | B-Track | C-Track |
| 1 | Assignment and Operations | 1, 2 | 3 | 4% | Week 2 | Week 3 | Week 3 |
| 2 | Functions and User Input | 3, 4 | 2 | 8% | Week 3 | Week 4 | Week 4 |
| 3 | Conditionals and Fruitful Functions | 5, 6 | 2 | 8% | Week 4 | Week 5 | Week 6 |
| 4 | Modules and Strings | 7, 8 | 2 | 8% | Week 5 | Week 6 | Week 8 |
| 5 | Files | 9 | 2 | 8% | Week 6 | Week 8 | Week 10 |
| 6 | Iteration: while loop | 10 | 2 | 8% | Week 8 | Week 10 | Week 12 |
| 7 | NumPy and Matrices | 11, 14-17 | 2 | 8% | Week 10 | Week 12 | Week 13 |
| 8 | Pandas and Visualisation | 18-19 | 2 | 8% | Week 12 | Week 13 | - |

Figure 1: The Mastery Levels. A/B/C-track are the expected time to finish a level if you want an A/B/C for the paper.

The first six Mastery Levels are foundational programming knowledge. These levels cover the basics of input, output, math, conditionals, and iteration, and are necessary for any understanding of programming. Everything else builds upon these concepts.

However, there are other topics that are also essential for effective programming, such as an understanding of data structures. In Python, the most important data structure is the list. All Mastery Levels higher than 6 assume some basic knowledge of lists.

Every student should sit at least one Mastery Level test per week. These tests will be held during the first half-hour of your scheduled laboratory session each week. If you want to take additional tests, you can attend additional lab sessions on different days.

The practical tests will be run under exam conditions in the lab. These tests will include questions similar to the Mastery Levels, but you will only have one chance to take each Practical Test. Both Practical Tests will have four questions, each worth 5% of the total grade. Practical Test 1, covering material from Lessons 1-8, is likely to be held in week 6. Practical Test 2, covering material from chapter 9 onwards, will be held in week 13.

There will be no final exam for this course.

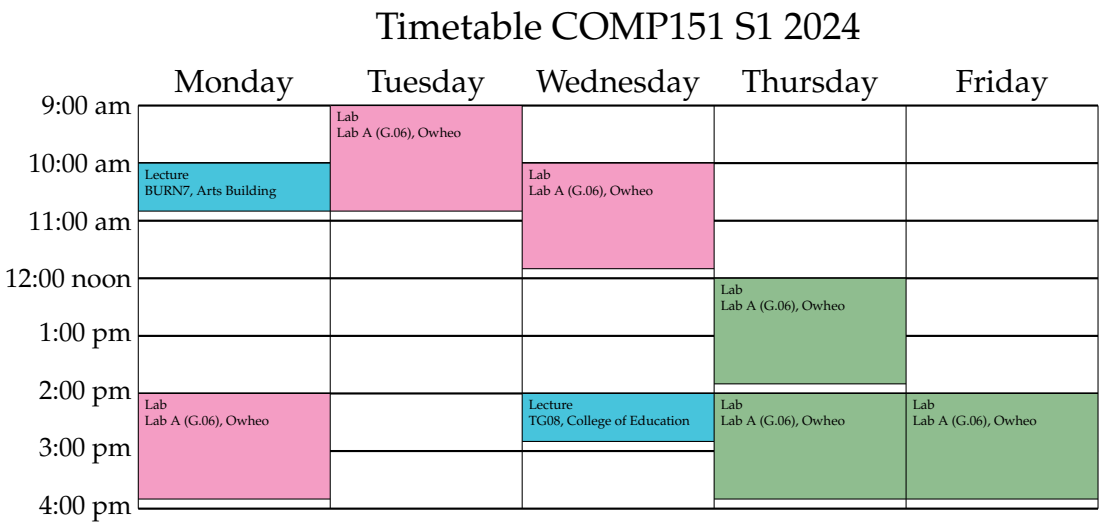
## Lectures and Laboratories

Lectures will be held on Monday at 10am and Wednesday at 2pm. These lectures will cover topics relevant to labs, mastery levels, practical tests, and programming in general.

Laboratories will be held in Lab A (G.06), in the Owheo Building (133 Union Street East). You will be assigned to two laboratory sessions per week, and each session will be two hours long. Your scheduled stream can be viewed on eVision: https://evision.otago.ac.nz/

**Attendance in laboratories is compulsory. Failure to attend labs will result in failure of the COMP151 course.**

You should attend your assigned lab time, and we will scan you into the lab. If there are available seats, you may also attend other lab sessions in addition to your assigned one.




Lab A is available to use 24/7. Please be respectful of the lab and the surrounding areas. Eating and drinking are discouraged inside the lab, there is a kitchen nearby you may use. If you do bring a drink, make sure it is in a spill-proof container. Additionally, we expect you to respect any other people who share the lab with you.

# Academic Integrity and Misconduct

In COMP151, all assessed work must be original and completed independently. You are welcome to discuss tasks and program design with others, but **the code submitted for assessment must be written by you** (not by someone else or generative tools). **Do not discuss the content the Mastery Tests or Practical Tests** with your peers. Attempting to save or share Test content is considered academic misconduct.

Other examples of Academic Misconduct include plagiarism, impersonation, unauthorised collaboration, accessing internet during test conditions, and aiding someone else's misconduct.

## Generative Tools

Unauthorised use of ChatGPT (or other generative tools) for your coursework is considered academic misconduct. While such tools may be helpful it is important to focus on building fundamental skills as a beginner. By relying on text-generating tools too early in your education, you risk impeding the development of important programming skills.

## Being Informed

It is your responsibility to be aware of and use acceptable academic practices. You should consult the Academic Integrity and Academic Misconduct (including A Brief Guide for Students), the policy on Academic Integrity, and the page on Plagiarism.

# Foreword

By David Beazley

As an educator, researcher, and book author, I am delighted to see the completion of this book. Python is a fun and extremely easy-to-use programming language that has steadily gained in popularity over the last few years. Developed over ten years ago by Guido van Rossum, Python's simple syntax and overall feel is largely derived from ABC, a teaching language that was developed in the 1980's. However, Python was also created to solve real problems and it borrows a wide variety of features from programming languages such as C++, Java, Modula-3, and Scheme. Because of this, one of Python's most remarkable features is its broad appeal to professional software developers, scientists, researchers, artists, and educators.

Despite Python's appeal to many different communities, you may still wonder why Python? or why teach programming with Python? Answering these questions is no simple task—especially when popular opinion is on the side of more masochistic alternatives such as C++ and Java. However, I think the most direct answer is that programming in Python is simply a lot of fun and more productive.

When I teach computer science courses, I want to cover important concepts in addition to making the material interesting and engaging to students. Unfortunately, there is a tendency for introductory programming courses to focus far too much attention on mathematical abstraction and for students to become frustrated with annoying problems related to low-level details of syntax, compilation, and the enforcement of seemingly arcane rules. Although such abstraction and formalism is important to professional software engineers and students who plan to continue their study of computer science, taking such an approach in an introductory course mostly succeeds in making computer science boring. When I teach a course, I don't want to have a room of uninspired students. I would much rather see them trying to solve interesting problems by exploring different ideas, taking unconventional approaches, breaking the rules, and learning from their mistakes. In doing so, I don't want to waste half of the semester trying to sort out obscure syntax problems, unintelligible compiler error messages, or the several hundred ways that a program might generate a general protection fault.

One of the reasons why I like Python is that it provides a really nice balance between the practical and the conceptual. Since Python is interpreted, beginners can pick up the language and start doing neat things almost immediately without getting lost in the problems of compilation and linking. Furthermore, Python comes with a large library of modules that can be used to do all sorts of tasks ranging from web-programming to graphics. Having such a practical focus is a great way to engage students and it allows them to complete significant projects. However, Python can also serve as an excellent foundation for introducing important computer science concepts. Since Python fully supports procedures and classes, students can be gradually introduced to topics such as procedural abstraction, data structures, and object-oriented programming—all of which are applicable to later courses on Java or C++. Python even borrows a number of features from functional programming languages and can be used to introduce concepts that would be covered in more detail in courses on Scheme and Lisp.

In reading Jeffrey's preface, I am struck by his comments that Python allowed him to see a higher level of success and a lower level of frustration and that he was able to move faster with better results. Although these comments refer to his introductory course, I sometimes use Python for these exact same reasons in advanced graduate level computer science courses at the University of Chicago. In these courses, I am constantly faced with the daunting task of covering a lot of difficult course material in a blistering nine week quarter. Although it is certainly possible for me to inflict a lot of pain and suffering by using a language like C++, I have often found this approach to be counterproductive—especially when the course is about a topic unrelated to just programming. I find that using Python allows me to better focus on the actual topic at hand while allowing students to complete substantial class projects.

Although Python is still a young and evolving language, I believe that it has a bright future in education. This book is an important step in that direction.

David Beazley University of Chicago Author of the Python Essential Reference

# Preface

By Jeffrey Elkner

This book owes its existence to the collaboration made possible by the Internet and the free software movement. Its three authors—a college professor, a high school teacher, and a professional programmer—have yet to meet face to face, but we have been able to work closely together and have been aided by many wonderful folks who have donated their time and energy to helping make this book better.

We think this book is a testament to the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

## How and why I came to use Python

In 1999, the College Board's Advanced Placement (AP) Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum at Yorktown High School in Arlington, Virginia, where I teach. Up to this point, Pascal was the language of instruction in both our first-year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first-year course for the 1997-98 school year so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first-year class, I went looking for an alternative to C++.

I needed a language that would run on the machines in our GNU/Linux lab as well as on the Windows and Macintosh platforms most students have at home. I wanted it to be free software, so that students could use it at home regardless of their income. I wanted a language that was used by professional programmers, and one that had an active developer community around it. It had to support both procedural and object-oriented programming. And most importantly, it had to be easy to learn and teach. When I investigated the choices with these goals in mind, Python stood out as the best candidate for the job.

I asked one of Yorktown's talented students, Matt Ahrens, to give Python a try. In two months he not only learned the language but wrote an application called pyTicket that enabled our staff to report technology problems via the Web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment, combined with Matt's positive assessment of Python, suggested that Python was the solution I was looking for.

## Finding a textbook

Having decided to use Python in both of my introductory computer science classes the following year, the most pressing problem was the lack of an available textbook.

Free documents came to the rescue. Earlier in the year, Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational materials. Allen had already written a first-year computer science textbook, *How to Think Like a Computer Scientist*. When I read this book, I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming rather than the features of a particular language. Reading it immediately made me a better teacher.

*How to Think Like a Computer Scientist* was not just an excellent book, but it had been released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version of the book into the new language. While I would not have been able to write a textbook on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative

development model used so well in software could also work for educational materials.

Working on this book for the last two years has been rewarding for both my students and me, and my students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for mistakes in the book by giving them a bonus point each time they made a suggestion that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object-oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book sat in an unfinished state for the better part of a year until the open source community once again provided the needed means for its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene, Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our Website at `http://openbookproject.net` called *Python for Fun* and was working with some of my most advanced students as a master teacher, guiding them beyond where I could take them.

# Introducing programming with Python

The process of translating and using *How to Think Like a Computer Scientist* for the past two years has confirmed Python's suitability for teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text illustrates this point. It is the traditional "hello, world program", which in the C++ version of the book looks like this:

```
#include <iostream.h>

void main()
{
   cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print("Hello, World!")
```

Even though this is a trivial example, the advantages of Python stand out. Yorktown's Computer Science I course has no prerequisites, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain the `#include`, `void main()`, `{`, and `}` statements and risk confusing or intimidating some of the students right at the start, or to tell them, "Just don't worry about all of that stuff now; we will talk about it later," and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to write their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing the explanatory text of the program in each version of the book further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world!" in the C++ version; in the Python version, there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high-level language like Python allows a teacher to postpone talking about low-level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically. One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place that holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus, the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both computer science and algebra). Bytes and addresses do not help the matter. In Python a variable is a name that refers to a thing. This is a far more intuitive concept for beginning students and is much closer to the meaning of "variable" that they learned in their math courses. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function

definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the keyword `def`, so I simply tell my students, "When you define a function, begin with `def`, followed by the name of the function that you are defining; when you call a function, simply call (type) out its name." Parameters go with definitions; arguments go with calls. There are no return types, parameter types, or reference and value parameters to get in the way, so I am now able to teach functions in less than half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years I taught C++. I move faster with better results. More students leave the course with the ability to create meaningful programs and with the positive attitude toward the experience of programming that this engenders.

# Building a community

I have received email from all over the globe from people using this book to learn or to teach programming. A user community has begun to emerge, and many people have been contributing to the project by sending in materials for the companion Website at `http://openbookproject.net/pybiblio`

With the continued growth of Python, I expect the growth in the user community to continue and accelerate. The emergence of this user community and the possibility it suggests for similar collaboration among educators have been the most exciting parts of working on this project for me. By working together, we can increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you. Please write to me at `jeff@elkner.net`.

Jeffrey Elkner
Arlington Public Schools
Arlington, Virginia

# Contributor List

To paraphrase the philosophy of the Free Software Foundation, this book is free like free speech, but not necessarily free like free pizza. It came about because of a collaboration that would not have been possible without the GNU Free Documentation License. So we would like to thank the Free Software Foundation for developing this license and, of course, making it available to us.

We would also like to thank the more than 100 sharp-eyed and thoughtful readers who have sent us suggestions and corrections over the past few years. In the spirit of free software, we decided to express our gratitude in the form of a contributor list. Unfortunately, this list is not complete, but we are doing our best to keep it up to date. It was also getting too large to include everyone who sends in a typo or two. You have our gratitude, and you have the personal satisfaction of making a book you found useful better for you and everyone else who uses it. New additions to the list for the 2nd edition will be those who have made on-going contributions.

If you have a chance to look through the list, you should realize that each person here has spared you and all subsequent readers from the confusion of a technical error or a less-than-transparent explanation, just by sending us a note.

Impossible as it may seem after so many corrections, there may still be errors in this book. If you should stumble across one, we hope you will take a minute to contact us. The email address is `jeff@elkner.net`. Substantial changes made due to your suggestions will add you to the next version of the contributor list (unless you ask to be omitted). Thank you!

## Otago Editions

- Sandy Garner for many many suggestions and improvements including many new exercises.
- Karen Gray for proof reading, exercises and suggestions.
- Special thanks to all the lab demonstrators in COMP151 that have made suggestions for improving the book.
- Raymond Scurr
- Michael Albert
- Matthew Jenkins
- Xiangfei Jia
- Thomas Harper
- Paul McCarthy

## Second Edition

- A special thanks to the students in Jeff's Computer Science class at the HB-Woodlawn program during the 2007-2008 school year: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail, and Iliana Vazuka. Your willingness to serve as beta-testers of the new chapters as they were completed, and to endure frequent changes in response to your feedback have proved invaluable. Thanks to you this is truly a *student tested* text.
- Ammar Nabulsi sent in numerous corrections from Chapters 1 and 2.
- Aldric Giacomoni pointed out an error in our definition of the Fibonacci sequence in Chapter 5.
- Roger Sperberg sent in several spelling corrections and pointed out a twisted piece of logic in Chapter 3.
- Adele Goldberg sat down with Jeff at PyCon 2007 and gave him a list of suggestions and corrections from throughout the book.
- Ben Bruno sent in corrections for chapters 4, 5, 6, and 7.
- Carl LaCombe pointed out that we incorrectly used the term "commutative" in chapter 6 where "symmetric" was the correct term.
- Alessandro Montanile sent in corrections for errors in the code examples and text in chapters 3, 12, 15, 17, 18, 19, and 20.

- Emanuele Rusconi found errors in chapters 4, 8, and 15.
- Michael Vogt reported an indentation error in an example in chapter 6, and sent in a suggestion for improving the clarity of the shell vs. script section in chapter 1.

# First Edition

- Lloyd Hugh Allen sent in a correction to Section 8.4.
- Yvon Boulianne sent in a correction of a semantic error in Chapter 5.
- Fred Bremmer submitted a correction in Section 2.1.
- Jonah Cohen wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML.
- Michael Conlon sent in a grammar correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters.
- Benoit Girard sent in a correction to a humorous mistake in Section 5.6.
- Courtney Gleason and Katherine Smith wrote `horsebet.py`, which was used as a case study in an earlier version of the book. Their program can now be found on the website.
- Lee Harr submitted more corrections than we have room to list here, and indeed he should be listed as one of the principal editors of the text.
- James Kaylin is a student using the text. He has submitted numerous corrections.
- David Kershaw fixed the broken `catTwice` function in Section 3.10.
- Eddie Lam has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme.
- Man-Yong Lee sent in a correction to the example code in Section 2.4.
- David Mayo pointed out that the word "unconsciously" in Chapter 1 needed to be changed to "subconsciously".
- Chris McAloon sent in several corrections to Sections 3.9 and 3.10.
- Matthew J. Moelter has been a long-time contributor who sent in numerous corrections and suggestions to the book.
- Simon Dicon Montford reported a missing function definition and several typos in Chapter 3. He also found errors in the `increment` function in Chapter 13.
- John Ouzts corrected the definition of "return value" in Chapter 3.
- Kevin Parks sent in valuable comments and suggestions as to how to improve the distribution of the book.
- David Pool sent in a typo in the glossary of Chapter 1, as well as kind words of encouragement.
- Michael Schmitt sent in a correction to the chapter on files and exceptions.
- Robin Shaw pointed out an error in Section 13.1, where the printTime function was used in an example without being defined.
- Paul Sleigh found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX.
- Craig T. Snydal is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections.
- Ian Thomas and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions.
- Keith Verheyden sent in a correction in Chapter 3.
- Peter Winstanley let us know about a longstanding error in our Latin in Chapter 3.
- Chris Wrobel made corrections to the code in the chapter on file I/O and exceptions.
- Moshe Zadka has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book.
- Christoph Zwerschke sent several corrections and pedagogic suggestions, and explained the difference between *gleich* and *selbe*.
- James Mayer sent us a whole slew of spelling and typographical errors, including two in the contributor list.
- Hayden McAfee caught a potentially confusing inconsistency between two examples.
- Angel Arnal is part of an international team of translators working on the Spanish version of the text. He has also found several errors in the English version.
- Tauhidul Hoque and Lex Berezhny created the illustrations in Chapter 1 and improved many of the other illustrations.

- Dr. Michele Alzetta caught an error in Chapter 8 and sent some interesting pedagogic comments and suggestions about Fibonacci and Old Maid.
- Andy Mitchell caught a typo in Chapter 1 and a broken example in Chapter 2.
- Kalin Harvey suggested a clarification in Chapter 7 and caught some typos.
- Christopher P. Smith caught several typos and is helping us prepare to update the book for Python 2.2.
- David Hutchins caught a typo in the Foreword.
- Gregor Lingl is teaching Python at a high school in Vienna, Austria. He is working on a German translation of the book, and he caught a couple of bad errors in Chapter 5.
- Julie Peters caught a typo in the Preface.

# Notes on the Otago Edition

This course book is a modified version of the online textbook: "How to think like a computer scientist: learning with Python, 2nd Edition", by Jeffrey Elkner, Allen B. Downey, and Chris Meyers. It has been modified for the Otago context and more recently for Python 3, and also the combined wisdom of the local course developers (Brendan McCane, Iain Hewson, Nick Meek and Karen Gray) have been distilled into written form. An online copy of this textbook is available from Blackboard.

This book is all you need for COMP151. There is no expensive textbook to buy. You can have an electronic copy for free, and either print it yourself at home, or view it on an electronic device of your choice (e.g. laptop, ipad etc).

We hope you enjoy COMP151: Programming for Scientists.

Brendan McCane
Karen Gray

# Lesson 1

# The way of the program

This paper has two goals: to teach you Python programming and to teach you to think like a **computer scientist**. Learning to think like a computer scientist is essential to becoming a competent programmer.

This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. Like scientists, they observe the behaviour of complex systems, form hypotheses, and test predictions.

This paper is perfect for those who want to gain practical programming skills to solve smallish programming problems. The single most important skill for a programmer is **problem solving**. Problem solving is the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. Learning to program is an excellent opportunity to improve your problem-solving skills. By learning to program, you will acquire a useful skill and use it as a means to an end, which will become clearer as you progress. That's why this chapter is called, "The way of the program."

## 1.1 The COMP151 lab

COMP151 labs are in Lab A (G06) on the ground floor of the Owheo Building (133 Union Street East). The lab has 28 computers running a recent version of Windows. You'll be assigned to 2 two-hour labs per week, and lab demonstrators will be available during those times to assist you. You can also use the lab at other times if there's an available seat, however if the lab becomes full you may be asked to use a different lab (likely Lab B). Check the whiteboard in the lab for details and hints.

To log in, use your standard university username and password, and don't forget to log out by pressing Ctrl+Alt+Delete and clicking Sign out when you leave the lab. See figure 1.1.

### 1.1.1 The Windows desktop

If you're already familiar with Apple macOS, you'll find Windows similar but with some differences. Basic actions like pointing, selecting, dragging-and-dropping, and left- and right-clicking with the mouse are much the same. Menus still exist in a familiar order of File, Edit, View, etc. The Taskbar replaces the Dock and it works slightly differently. The following tips will help you navigate Windows for use in these labs.

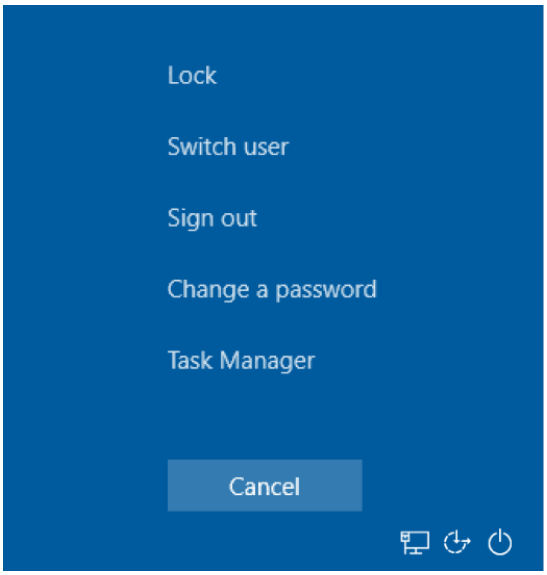



Figure 1.1: First way to Sign out.
Press Ctrl+Alt+Delete then click Sign out

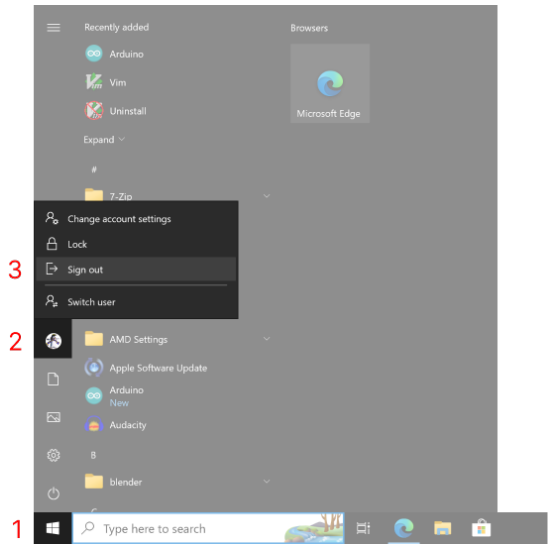



Figure 1.2: Another way to Sign out.
Start menu → User icon → Sign out

### 1.1.2 The Start Menu

The Start menu is found on the taskbar and it provides access to all of our applications and settings. To open the Start menu, simply click on the Windows icon in the lower-left corner of the screen.

The 3 main applications we will use in this course are *Thonny*, *File Explorer*, and a *Web Browser*. Thonny is the most important and will be used to create Python scripts and programs.

Figure 1.3: Thonny icon

Figure 1.4: File Explorer

Figure 1.5: Web Browsers
Edge, Firefox, and Chrome

To find Thonny, use the Start menu, or the Search Bar (to the right of the Start menu). Once you have located Thonny you could pin it to the taskbar for easy access later, see figure 1.6.

**Note:** You can complete coursework at home if you install Thonny, which is free for Windows, macOS, and Linux. If you need help installing Thonny on your own computer ask a demonstrator for assistance.

### 1.1.3 File Explorer

File Explorer allows you to manage the files and folders on your computer. It is much the same as Finder on macOS, it can create, copy, move, rename, and delete files and folders.

To open File Explorer, you can either click the folder icon in the taskbar (See figure 1.4), or press the Windows key + E on your keyboard. Once opened, the left column is the navigation pane listing places, drives and locations, and on the right the main window displays the files and folders as icons.

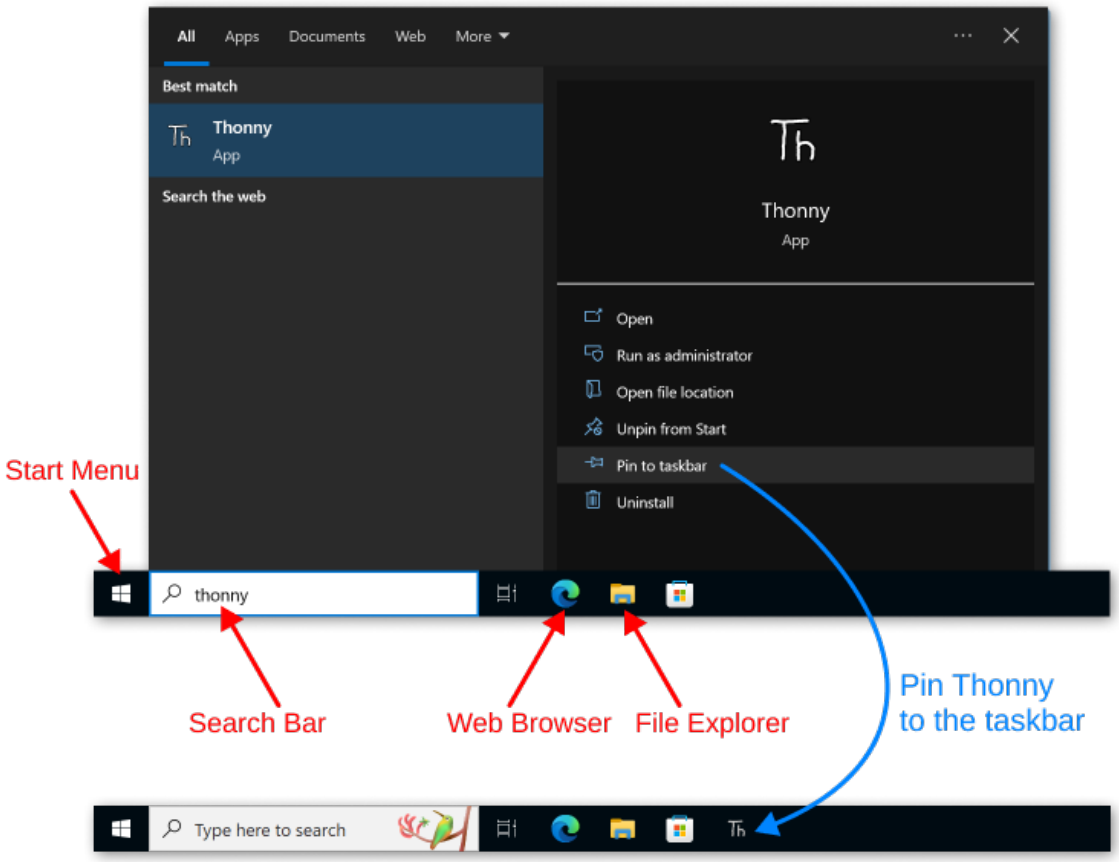



Figure 1.6: Pinning Thonny to the Windows taskbar

## Your home directory

The lab computers use network synced file storage. This means that your files are stored remotely and can be accessed from any computer. The **Home Drive H:** automatically syncs your *Desktop*, *Documents*, *Downloads* and other folders, making them easily accessible when switching between computers.

However, your **Home Drive H:** has limited storage space, which is why we recommend you store extra files on your **OneDrive**. Files you are currently working on belong in your **H: Drive**, files for long term storage belong in your **OneDrive**.

Another advantage of OneDrive is that you can access your files from your own computer (Windows/-Mac/Linux), and even through your StudentMail. More information about the University of Otago's OneDrive can be found here.

## Tidy Files

Most of the files in this course are Python scripts, and its a good idea to keep these files neat and tidy. The best way to do this is to have separate folder for each lab. Using File Explorer navigate to `Documents`, create a folder called `COMP151` and inside that folder create another called `Lab1` (next lab create `Lab2`, ...). You are going to be creating a lot of files, keeping them organised will make your life easier.

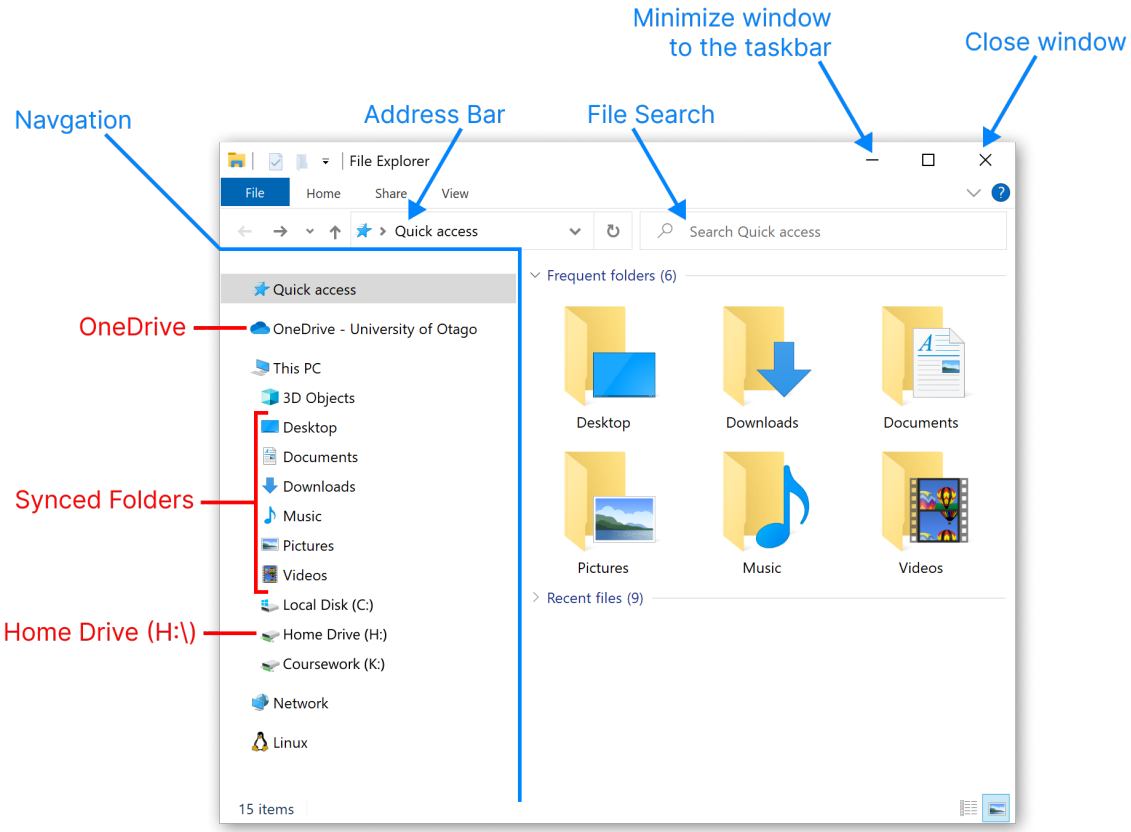



Figure 1.7: File Explorer Window

## 1.2 Blackboard

Many papers at Otago make use of Blackboard, an online course management tool that provides staff and students with communication and file sharing tools as well as a host of other features. COMP151 will use Blackboard for making class announcements, and sharing material. You should check Blackboard every time you come to the lab. To log-on to Blackboard:

- Open a web browser (Edge, Safari, Firefox) and navigate to `http://blackboard.otago.ac.nz`.
- Log-on (using your *University* username and password).
- Locate the `MyPapers` group of links and click on the `COMP151_S1DNI_2023: Programming for Scientists` link. You will be taken to the Announcements page. This is where all important course-related messages will be posted.
- Read the announcements.
- Examine the options you have on the left.

### 1.2.1 Getting help

In the lab, demonstrators will be available to assist you, but before you seek their help, consider if the answer to your question is already in the book, if you have tried various approaches, if you know precisely

what the problem is, or if you have searched the internet or asked the person next to you. We encourage you to develop problem-solving skills independently during this semester. However, if you are genuinely stuck, do not hesitate to ask for help.


Figure 1.8: DemCall.
It has the same icon as the Web Browser, Edge

To request assistance open "DemCall" from the desktop (see figure 1.8). You must provide your username, which paper, and which lab you are in (Lab A). Clicking "Raise Hand" will add you to the queue. A demonstrator will be with you shortly. While waiting, continue working on the problem. If you solve it, you can remove yourself from the queue by clicking "Cancel".

### 1.2.2 Coursework files

For some exercises in this course we will supply you with files to work with. These files can be found on Blackboard under `Course Files`. At the start of each lab copy the appropriate folder to your Desktop or Home drive by simply clicking and dragging the appropriate folder. There is a folder for this lab. Try dragging `Test file` onto your desktop from `Lab1/Coursefiles`.

## 1.3 Thonny

In COMP151 you will be using a development environment (also called an **IDE** - Integrated Development Environment) called **Thonny**. Thonny is a very simple IDE and is ideal for our purposes. Start Thonny by clicking on the Thonny shortcut icon in the taskbar that you created moments ago. You should see a window that looks like Figure 1.9. By default Thonny starts with an interpreter window open on the bottom and an editor window for scripts at the top.. Click on the interpreter window to give it *focus*.

## 1.4 The Python programming language

The programming language you will be learning is Python, specifically Python 3. Python is an example of a **high-level language**; other high-level languages you might have heard of are C++, PHP, and Java.

As you might infer from the name high-level language, there are also **low-level languages**, sometimes referred to as machine languages or assembly languages. Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds

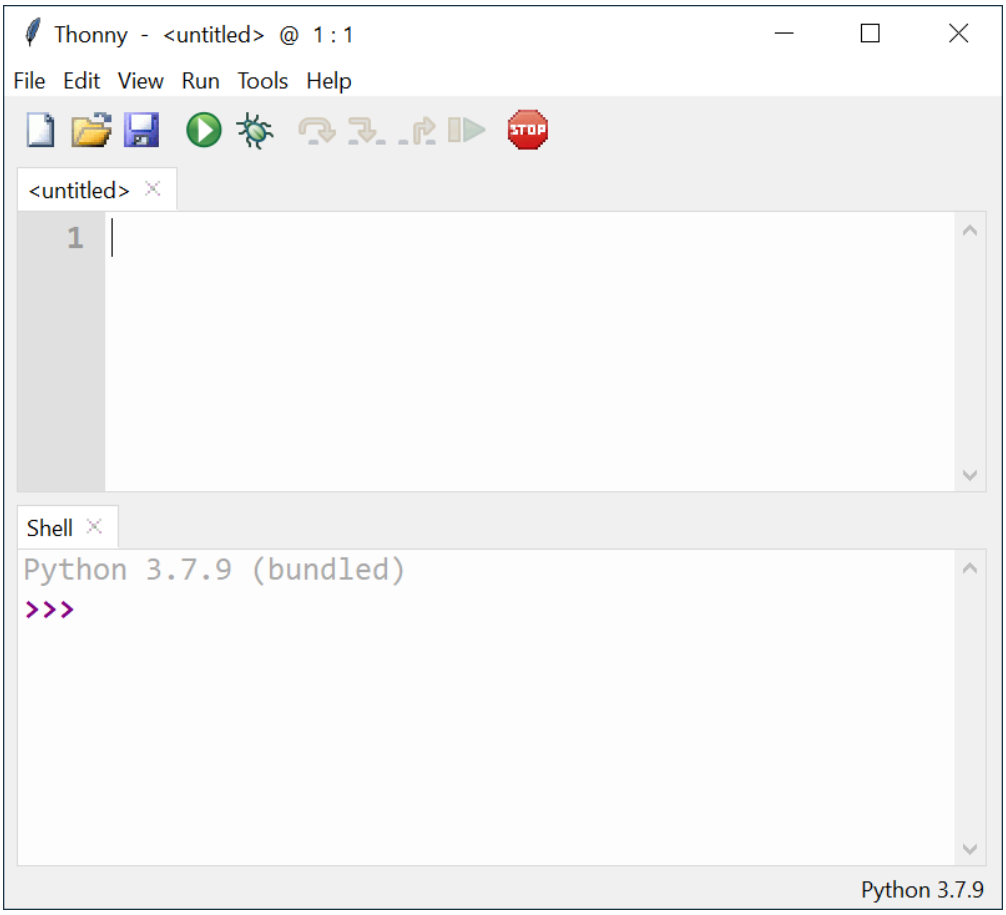



Figure 1.9: Thonny starts with both a script editor and an interpreter window open.

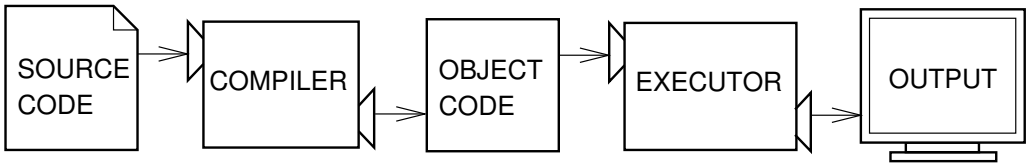
of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications.

Two kinds of applications process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.


A compiler reads the program and translates it into a low-level program, which can then be run.

In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.




Many modern languages use both processes. They are first compiled into a lower level language, called **byte code**, and then interpreted by a program called a **virtual machine**. Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language.

There are two ways to use the Python interpreter: *shell mode* and *script mode*. In shell mode, you type Python statements into the **Python shell** and the interpreter immediately prints the result.

As mentioned earlier we will be using an IDE (Integrated Development Environment) called **Thonny**. When you first start Thonny it will open a window with two sections. The top one is a text editor where you can type and edit **scripts**. The interpreter/ shell is on the bottom. Let us begin with the shell.

```
Python 3.6.1
>>>
```

The first line identifies the version of Python being used. The last line starts with >>>, which is the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions.

If we type `print(1 + 1)` the interpreter will reply `2` and give us another prompt.

```
>>> print(1 + 1)
2
>>>
```

Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a **script**. For example, we used the text editor in Thonny (but we could have used any text editor) to create a file named `first_program.py`, with the following contents:

```
print(1 + 1)
```

By convention, files that contain Python programs have names that end with `.py`.

Thonny simplifies the whole process by presenting the interpreter window and a text editor within the same application. The easiest way to run a script in Thonny is to open the script in the top window and press the arrow. The script's output will be displayed in the shell.

Most programs are more interesting than this one. The examples in this book use both the Python interpreter and scripts. You will be able to tell which is intended since shell mode examples (i.e. entering lines directly into the interpreter) will always start with the Python prompt, >>>. Working in shell mode is convenient for testing short bits of code because you get immediate feedback. Think of it as scratch paper used to help you work out problems. Anything longer than a few lines should be put into a script so it can be saved for future use.

## 1.5 What is a program?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or (strangely enough) compiling a program.

The details look different in different languages, but a few basic instructions appear in just about every language:

**input:** Get data from the keyboard, a file, or some other device.

**output:** Display data on the screen or send data to a file or other device.

**math:** Perform basic mathematical operations like addition and multiplication.

**conditional execution:** Check for certain conditions and execute the appropriate sequence of statements.

**repetition:** Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look more or less like these. Thus, we can describe programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

That may be a little vague, but we will come back to this topic later when we talk about **algorithms**.

## 1.6 What is debugging?

Programming is a complex process, and because it is done by human beings, programs often contain errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

**Syntax errors** Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a

single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer syntax errors and find them faster.

**Runtime errors** The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

**Semantic errors** The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

## 1.7 Debugging in Thonny

The development environment that we will be using, Thonny, offers several tools to help you debug your programs. It includes a debugger that allows you to run your program step by step. A short video showing how this might be useful is available on Blackboard. You can also read more about Thonny and its many useful features at `http://thonny.org/`.

## 1.8 Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is usually a trial and error process. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (A. Conan Doyle, *The Sign of Four* )

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, one of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved into Linux (*The Linux Users' Guide* Beta Version 1).

Later chapters will make more suggestions about debugging and other programming practices.

## 1.9 Formal and natural languages

**Natural languages** are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

**Formal languages** are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

*Programming languages are formal languages that have been designed to express computations.*

Formal languages tend to have strict rules about syntax. For example, 3+3=6 is a syntactically correct mathematical statement, but 3=+6$ is not. $H_2O$ is a syntactically correct chemical name, but $_2Zz$ is not. Syntax rules come in two flavors, pertaining to **tokens** and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3=+6$ is that $ is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz. The second type of syntax rule pertains to the structure of a statement—that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called **parsing**. For example, when you hear the sentence, "The brown dog barked", you understand that the brown dog is the subject and barked is the verb. Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a dog is and what it means to bark, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many differences:

**ambiguity:** Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

**redundancy:** In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

**literalness:** Natural languages are full of idiom and metaphor. If someone says, "The penny dropped", there is probably no penny and nothing dropped. Formal languages mean exactly what they say.

People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:

**Poetry:** Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

**Prose:** The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

**Programs:** The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

## 1.10 The first program

Traditionally, the first program written in a new language is called Hello, World! because all it does is display the words, Hello, World! We're going to use Kia ora te ao instead. In Python, it looks like this:

```
print("Kia ora te ao!")
```

This is an example of calling the **print function**, which doesn't actually print anything on paper. It displays a value on the screen. In this case, the result is the words

```
Kia ora te ao!
```

The quotation marks in the program mark the beginning and end of the value; they don't appear in the result.

Some people judge the quality of a programming language by the simplicity of the Kia ora te ao program. By this standard, Python does about as well as is possible. Writing this program is one of the exercises described at the end of this lesson.

## 1.11 Glossary

**algorithm:** A general process for solving a category of problems.

**bug:** An error in a program.

**byte code:** An intermediate language between source code and object code. Many modern languages first compile source code into byte code and then interpret the byte code with a program called a *virtual machine*.

**compile:** To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

**debugging:** The process of finding and removing any of the three kinds of programming errors.

**exception:** Another name for a runtime error.

**executable:** Another name for object code that is ready to be executed.

**formal language:** Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

**high-level language:** A programming language like Python that is designed to be easy for humans to read and write.

**interpret:** To execute a program in a high-level language by translating it one line at a time.

**low-level language:** A programming language that is designed to be easy for a computer to execute; also called machine language or assembly language.

**natural language:** Any one of the languages that people speak that evolved naturally.

**object code:** The output of the compiler after it translates the program.

**parse:** To examine a program and analyze the syntactic structure.

**portability:** A property of a program that can run on more than one kind of computer.

**print function:** A function that causes the Python interpreter to display a value on the screen.

**problem solving:** The process of analysing a problem, finding a solution, and expressing the solution.

**program:** a sequence of instructions that specifies to a computer actions and computations to be performed.

**Python shell:** An interactive user interface to the Python interpreter. The user of a Python shell types commands at the prompt (>>>), and presses the return key to send these commands immediately to the interpreter for processing.

**runtime error:** An error that does not occur until the program has started to execute but that prevents the program from continuing.

**script:** A program stored in a file (usually one that will be interpreted).

**semantic error:** An error in a program that makes it do something other than what the programmer intended.

**semantics:** The meaning of a program.

**source code:** A program in a high-level language before being compiled.

**syntax:** The structure of a program.

**syntax error:** An error in a program that makes it impossible to parse (and therefore impossible to interpret).

**token:** One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

## 1.12 Laboratory exercises

### Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

1. "this is a syntactically incorrect sentence" has two syntax errors. What are they?

2. Write an English sentence with understandable semantics but incorrect syntax. If English is not your first language, feel free to use the language of your choice for this question and the next.

3. "Three of his legs were both the same." is an example of a sentence with correct syntax but some semantic errors, that is it is not clear what it means. Think up another sentence that is syntactically ok but has semantic (meaning) errors.

4. If you haven't already done so, start **Thonny**. Type `2 + 3` at the prompt (in the bottom section) and then hit return. The symbols + and − stand for plus and minus as they do in mathematics. The asterisk ($\star$) is the symbol for multiplication, and `**` is the symbol for exponentiation (so `5**2` means 5 to the power of 2). Python *evaluates* any *expression* you type in, prints the result, and then prints another prompt. Experiment by entering different expressions and recording what is printed by the Python interpreter.

.

| | Expression | Answer |
|---|---|---|
| 1. | | |
| 2. | | |
| 3. | | |
| 4. | | |
| 5. | | |
| 6. | | |
| 7. | | |

**Show your work to a demonstrator here to make sure you are on the right track.**

5. Type `print("hello")` at the prompt. Python executes this statement, which has the effect of printing the letters h-e-l-l-o. Notice that the quotation marks that you used to enclose the string are not part of the output.

   Now type `print('"hello"')` and describe and explain your result.

6. In the script editor window of Thonny, which is currently called <untitled>, type `print("hello")`. You can save your script with *File* → *Save* OR using the *save icon* above the script editor OR typing *Command-s*). When the save dialog opens you should make sure you are in your home directory before creating a New Folder called `Lab1`, and finally changing the name *untitled* to `hello.py` before clicking Save. Once you have done this run it by either pushing *F5* OR clicking *Run* → *Run current script* OR clicking the *arrow icon* above the script editor. Write down and explain what happens in the prompt window below:

7. Type `print(cheese)` without the quotation marks into the interpreter. The output will look something like this:

```
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'cheese' is not defined
```

   This is a run-time error; specifically, it is a NameError, and even more specifically, it is an error because the name cheese is not defined. If you don't know what that means yet, you will soon.

8. Still in the interpreter, type `1 2` (with a space in between) and then hit return. Python tries to evaluate the expression, but it can't because the expression is not syntactically legal. Instead, it prints the error message:

```
  File "<pyshell>", line 1
    1 2
      ^
SyntaxError: invalid syntax
```

   In many cases, Python indicates where the syntax error occurred, but it is not always right, and it doesn't give you much information about what is wrong. So, for the most part, the burden is on you to learn the syntax rules. In this case, Python is complaining because there is no operator between the numbers.

   Write down three more examples of statements that will produce error messages when you enter them at the Python prompt. Explain why each example is not valid Python syntax.

9. Whenever an *expression* is typed at the Python prompt, it is *evaluated* and the result is printed on the line below. `"This is a test..."` is an expression, which evaluates to `"This is a test..."` (just like the expression 42 evaluates to 42). In a script, however, evaluations of expressions are not sent to the program output, so it is necessary to explicitly print it.
   Type `"This is a test..."` at the prompt and hit enter. Record what happens.

10. Now open a new file (*File* → *New*, OR use the *New icon* above the script editor OR type *Command-N*) and type `"This is a test..."` into the Untitled document.

    Save your file in the `Lab1` folder (you can quickly change to it using the drop down list) as `test.py`. What happens when you run this script (the *arrow icon* OR *Run* → *Run current script*, OR *F5*)?

    Now change the contents to:

```
print("This is a test...")
```

    save and run it again. What happened this time?

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 2

# Variables, expressions and statements

## 2.1 Values and types

A **value** is one of the fundamental things—like a letter or a number—that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), and "Kia or te ao!". These values belong to different **classes**: 2 is an **int** (the class for integers or whole numbers) , and "Kia or te ao!" is a **str** (the class for strings). You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print function also works for integers.

```
>>> print(4)
4
```

If you are not sure what class a value belongs to, the interpreter can tell you.

```
>>> type("Kia ora te ao!")
<class 'str'>
>>> type(17)
<class 'int'>
```

For now the words `type` and `class` may be used interchangeably.

Strings belong to the class **str** and integers belong to the class **int**. Less obviously, numbers with a decimal point belong to a class called **float**, because these numbers are represented in a format called *floating-point*.

```
>>> type(3.2)
<class 'float'>
```

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

```
>>> type("17")
<class 'str'>
>>> type("3.2")
<class 'str'>
```

They're strings. Strings in Python can be enclosed in either single quotes (') or double quotes ("):

```
>>> type('This is a string.')
```

```
<class 'str'>
>>> type("And so is this.")
<class 'str'>
```

Double quoted strings can contain single quotes inside them, as in `"What's your name?"`, and single quoted strings can have double quotes inside them, as in `'The cat said "Meow!"'`.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. Resist the temptation as the commas mean something completely different in Python... which we will look at later in the course.

## 2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value. The **assignment statement** creates new variables and assigns them values:

```
>>> message = "What's your name?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string `"What's your name?"` to a new variable named `message`. The second assigns the integer 17 to `n`, and the third assigns the floating-point number 3.14159 to `pi`. The **assignment operator**, =, should not be confused with an equals sign (even though it uses the same character). Assignment operators link a *name*, on the left hand side of the operator, with a *value*, on the right hand side. This is why you will get an error if you enter:

```
>>> 17 = n
```

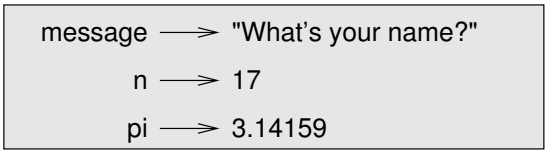
We strongly recommend that you read the assignment operator as "gets the value of".

For

```
>>> n = 17
```

we would read **n gets the value of 17**.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



The print function also works with variables.

```
>>> print(message)
What's your name?
>>> print(n)
17
>>> print(pi)
3.14159
```

In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

The type of a variable is the type (or kind) of value it refers to.

## 2.3 Variable names and keywords

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. **Variable names** can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. `Bruce` and `bruce` are different variables. The underscore character (_) can appear in a name. It is often used in names with multiple words, such as `my_name` or `price_of_tea_in_china`. If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
  File "<pyshell>", line 1
    76trombones = "big parade"
               ^
SyntaxError: invalid syntax
>>> more$ = 1000000
  File "<pyshell>", line 1
    more$ = 1000000
         ^
SyntaxError: invalid syntax
>>> class = "COMP151"

  File "<pyshell>", line 1
    class = "COMP151"
          ^
SyntaxError: invalid syntax
```

`76trombones` is illegal because it does not begin with a letter. `more$` is illegal because it contains an illegal character, the dollar sign. But what's wrong with `class`? It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.

Python has thirty-five keywords:

```
False       None        True        and         as
assert      async       await       break       class
continue    def         del         elif        else
except      finally     for         from        global
if          import      in          is          lambda
nonlocal    not         or          pass        raise
return      try         while       with        yield
```

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

## 2.4 Statements

A **statement** is an instruction that the Python interpreter can execute. When you type a statement into the shell, Python executes it and displays the result, if there is one. Assignment statements don't produce a visible result. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute. For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

Again, the assignment statement produces no output.

## 2.5 Evaluating expressions

An **expression** is a combination of values, variables, and operators that represents a single result value. If you type an expression on the command line, the interpreter **evaluates** it and displays the result:

```
>>> 1 + 1
2
```

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

```
>>> 17
17
>>> x
2
```

Confusingly, evaluating an expression is not quite the same thing as printing a value.

```
>>> message = "What's your name?"
>>> message
"What's your name?"
>>> print(message)
What's your name?
```

When the Python shell displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But the print function prints the value of the expression, which in this case is the contents of the string.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
"Kia ora te ao!"
1 + 1
```

produces no output at all. How would you change the script to display the values of these four expressions?

## 2.6 Operators and operands

**Operators** are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**. The following are all legal Python expressions whose meaning is more or less clear:

```
20 + 32
hour - 1
hour * 60 + minute
minute / 60
5 ** 2
(5 + 9) * (15 - 7)
```

The symbols `+`, `-`, `*`, `/`, have the usual mathematical meanings. The symbol, `**`, is the exponentiation operator. The statement, `5 ** 2`, means 5 to the power of 2, or 5 squared in this case. Python also uses parentheses for grouping, so we can force operations to be done in a certain order just like we can in mathematics.

When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

## 2.7 Integer division

Python offers another kind of division called integer division. You may possibly remember it as the kind of division you did when you were very little and didn't know about decimal places or fractions.

It is represented with the operator `//` and produces the number of times the second operand **completely** fits into the first. Any remainder is discarded.

```
>>> 100 // 10
10
>>> 101 // 10
10
>>> 7 // 10
0
```

When one or both operands is a `float` the operator still produces the number of times the second operand **completely** fits into the first but the result is a `float`. Make sure you understand the examples below:

```
>>> 7.0 // 3
2.0
>>> 7.0 // 3.0
2.0
>>> 7.6 // 3
2.0
>>>
```

## 2.8 The modulus operator

The **modulus operator** works on numbers (and numerical expressions) and produces the remainder when the first operand is divided by the second using int division. In Python, the modulus operator is a percent sign (`%`). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
>>> 7.5 % 1.5
0.0
>>> 8.0 % 1.5
0.5
>>> 8.0 % 3
2.0
```

So, using int division, `7` divided by `3` is `2` with `1` left over. The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by (i.e. a multiple of) `y`. Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

Note that, again, if one or both of the operands is a `float` then the result will be also.

## 2.9 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym PEMDAS is a useful way to remember the order of operations:

1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2*(3-1)` is `4`, and `(1+1)**(5-2)` is 8. You can also use parentheses to make an expression easier to read, as in `(59*100)//60`, even though it doesn't change the result.

2. **E**xponentiation has the next highest precedence, so `2**1+1` is `3` and not `4`, and `3*1**3` is `3` and not `27`.

3. **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So `2*3-1` yields `5` rather than `4`, and `2//3-1` is `-1`, not `1` (remember that in integer division, `2//3=0`).

Operators with the same precedence are evaluated from left to right. So in the expression `59*100//60` the multiplication happens first, yielding `5900//60`, which in turn yields `98`. If the operations had been evaluated from right to left, the result would have been `59*1`, which is `59`, which is radically different! Similarly, in evaluating `17-4-3`, `17-4` is evaluated first.

If in doubt, use parentheses.

## 2.10 Operations on strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

```
message - 1
"Kia ora" / 123
message * "Kia ora"
"15" + 2
```

Interestingly, the + operator does work with strings, although it might not do exactly what you expect. For strings, the + operator represents **concatenation**, which means joining the two operands by linking them end-to-end. For example:

```
fruit = "banana"
baked_good = " nut bread"
print(fruit + baked_good)
```

The output of this program is `banana nut bread`. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings. The * operator also works on strings; it performs repetition. For example, `"Fun"*3` is `"FunFunFun"`. One of the operands has to be a string; the other has to be an integer. On one hand, this interpretation of + and * makes sense by analogy with addition and multiplication. Just as `4*3` is equivalent to `4+4+4`, we expect `"Fun"*3` to be the same as `"Fun"+"Fun"+"Fun"`, and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

## 2.11 Glossary

**value:** A number or string (or other thing to be named later) that can be stored in a variable or computed in an expression.

**type:** A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (type int), floating-point numbers (type float), and strings (type string).

**type function:** A function which returns the class of the given argument, for example:

```
>>> type("Kia ora te ao!")
<class 'str'>
>>> type(17)
<class 'int'>
```

**int:** A Python data type that holds positive and negative whole numbers.

**str:** A Python data type that holds a string of characters.

**float:** A Python data type which stores *floating-point* numbers. Floating-point numbers are stored internally in two parts: a *base* and an *exponent*. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use floats, and remember that they are only approximate values.

**variable:** A name that refers to a value.

**assignment statement:** A statement that assigns a value to a name (variable). To the left of the assignment operator, =, is a name. To the right of the assignment operator is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the following assignment:

```
n = n + 1
```

n plays a very different role on each side of the =. On the right it is a *value* and makes up part of the *expression* which will be evaluated by the Python interpreter before assigning it to the name on the left.

**assignment operator:** = is Python's assignment operator, which should not be confused with the mathematical comparison operator using the same symbol.

**state diagram:** A graphical representation of a set of variables and the values to which they refer.

**variable name:** A name given to a variable. Variable names in Python consist of a sequence of letters (a..z, A..Z, and _) and digits (0..9) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program *self documenting*.

**keyword:** A reserved word that is used by the compiler to parse a program; you cannot use keywords like `if`, `def`, and `while` as variable names.

**statement:** An instruction that the Python interpreter can execute. Examples of statements include the assignment statement and expression statements.

**expression:** A combination of variables, operators, and values that represents a single result value.

**evaluate:** To simplify an expression by performing the operations in order to yield a single value.

**operator:** A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

**operand:** One of the values on which an operator operates.

**modulus operator:** An operator, denoted with a percent sign (%), that works on integers and floats and yields the remainder when one number is divided by another.

**integer division:** An operation that divides one number by another. Integer division yields only the whole number of times that the numerator is divisible by the denominator and discards any remainder. If both operands are integers, so is the result. If one or both operands are floats, then so is the result.

**rules of precedence:** The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

**concatenate:** To join two operands end-to-end.

## 2.12 Laboratory exercises

1. Create a Folder in your COMP151 Directory called `Lab2` to store the files you create during this lab.

2. Enter the following statements into the interpreter and note which ones produce an error:

```
>>> n = 7
>>> 7 = n
>>> n = 5 + 2
>>> n = 5 + n
>>> n + 5 = n
>>> n = y
```

3. What are variables used for?

4. Enter the following calls to the print function into the interpreter and note which ones produce an error:

```
>>> print(n)
>>> print(n + 5)
>>> print(n + 5 + 2)
>>> print(n = 5)
>>> print(7 + 5)
>>> print(n, 7+5, n+7)
```

5. What can be passed into the print function?

6. Look at the following lines of code and predict what will be printed out. Use a python script to check your prediction.

```
n = 2
print(n + 2)
print("n is ", n)

n = 2
n = n + 2
print("n is ", n)
```

7. Take the sentence: *Jack is a dull boy.* Show how you would use the interpreter to store each word in a separate variable, then print out the sentence on one line using the print function.

   Check that your answer is correct by entering it into a script (a script is another name for a saved Python file) called jack.py and running it.

   **Show your work to a demonstrator here to make sure you are on the right track.**

8. Add parentheses to the expression `6*1-2` to change its value from `4` to `-6`.

9. Try to evaluate the following numerical expressions in your head, and put the answer beside each one, then use the Python interpreter to check your results:

```
>>> 5 % 2
>>> 9 % 5
>>> 15 % 12
>>> 12 % 15
>>> 6 % 6
>>> 9.6 % 3
>>> 0 % 7
>>> 7 % 0
```

   What happened with the last example? Why? If you were able to correctly anticipate the computer's response in all but the last one, it is time to move on. If not, take time now to make up examples of your own. Explore the modulus operator until you are confident you understand how it works.

10. Start the Python interpreter and enter `bruce + 4` at the prompt. This will give you an error:

```
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'bruce' is not defined
```

    Assign a value to `bruce` so that `bruce + 4` evaluates to `10`.

11. In a python script, assign a value greater than 1000 to the variable `x` e.g. 1468. Write code which will assign to a variable `xrounded` the value of `x` rounded down to the nearest 100 e.g. for x of 1468, xrounded will be 1400. Your code should display both values. Note that your code should work for *any* value of `x`.

    There are a couple of different ways of doing this using the operators discussed in this chapter.

## 2.13 Mastery Level 1 - Practice Questions

Mastery Level 1 tests your understanding of material covered in lessons 1 and 2.

There are 3 questions in the test.

You should check your solutions with a demonstrator.

**Part 1 - Expressions**

1. Given the following:

   ```
   s1 = "friends "
   s2 = "for "
   s3 = "ever "
   ```

   write an expression using only these variables along with suitable operators that will evaluate to a single string:

   ```
   "for ever friends friends for ever "
   ```

   Note the spaces that are included in the strings.

   In the actual test you would need to show a demonstrator that it works.

2. Given the following:

   ```
   s1 = "It's "
   s2 = "it's "
   s3 = "goodnight from "
   s4 = "him "
   s5 = "and "
   s6 = "me"
   ```

   write an expression using only these variables, a quoted full stop and suitable operators that will evaluate to a single string:

   ```
   "It's goodnight from him and it's goodnight from me."
   ```

   Note the spaces that are included in the strings.

   In the actual test you would need to show a demonstrator that it works.

3. Given the following:

   ```
   x = 19
   y = 7
   z = 3
   ```

   Write an expression using only these variables (and any operators that you need) that will evaluate to 119.

   Each variable must be used at least once and there are many possible answers.

   In the actual test you would need to show a demonstrator that it works.

4. Given the following:

```
x = 200
y = 4
z = 3
```

   Write an expression using only these variables (and any operators that you need) that will evaluate to 84.

   Each variable must be used at least once and there are many possible answers.

   In the actual test you would need to show a demonstrator that it works.

5. Complete the assignment statements:

```
x =
y =
z =
```

   So that

```
print(x ** 2 + 2 * y + z)
```

   displays 140. (There are infinitely many possible answers)

6. Complete the assignment statements:

```
x =
y =
z =
```

   So that

```
print(z//(x+y))
```

   displays 90. (There are infinitely many possible answers)

**Part 2 - Scripts**

1. Enter the following code into a Python script:

```
x = 3
y = 4.4
```

   Add code to print the product of x and y on one line and the type of data resulting from this expression on another.

2. Enter the following code into a Python script:

```
word_1 = "Good"
word_2 = "dog"
p_1 = "!"
```

   Add to the script so that it uses the variables above to print:

```
"Good dog!  Good dog!  Good dog!  "
```

   Note no spaces are included in the strings. You may include any further quoted spaces you require.

3. Enter the following code into a Python script:

```
s1 = "Ta"
s2 = "ra"
s3 = "boom de ay"
s4 = "!"
```

   Add to the script so that it uses the variables above to print:

```
"Ta ra ra boom de ay!!!"
```

   Note any spaces that are included in the strings. You may include any further quoted spaces you require.

4. Enter the following code into a Python script:

```
x = 3
y = "Bang!"
```

   Add to the script so that y is printed x times on one line with no added spaces.

**Part 3 - Scripts with numerical expressions**

1. Enter the following code into a Python script:

```
x = 47
y = 10
```

   Add code to print the remainder when x is divided by y

2. Enter the following code into a Python script:

```
x = 3
y = 5
z = 9
```

   Add code to print the result of x raised to the power of y, then divided by z using int division

3. Enter the following code into a Python script:

```
x = 3
y = 15
z = 9
```

   Add code to print x raised to the power of the result of y divided by z

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 3

# Functions: part 1

## 3.1 Composing expressions

So far, we have looked at the elements of a program—variables, expressions, and statements—in isolation, without talking about how to combine them. One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

```
>>>  print(17 + 3)
20
```

In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time. The point is that any valid expression involving literal values and variables (and other things we haven't yet covered) can be *passed in* (placed inside the parentheses) when we call the `print` function:

```
hour = 2
minute = 45
print(hour * 60 + minute)
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
percentage = (minute * 100) / 60
```

This ability may not seem impressive now, but you will see other examples where composition makes it possible to express complex computations neatly and concisely.

**Warning:** There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So, the following is illegal:

```
>>> minute + 1 = hour
  File "<pyshell>", line 1
SyntaxError: can't assign to operator
```

## 3.2 Function calls

Many common tasks come up time and time again when programming. Instead of requiring you to constantly reinvent the wheel, Python has a number of built-in features which you can use. Including so much ready to use code is sometimes referred to as a 'batteries included' philosophy. Python comes with nearly seventy predefined functions (of which we'll be only using about a dozen) and the simplest way to use this prewritten code is via function calls.[1]

The syntax of a function call is simply

```
FUNCTION_NAME(ARGUMENTS)
```

Not all functions take an argument, and some take more than one (in which case the arguments are separated by commas).

In addition to `print` you have already seen another example of a **function call**:

```
>>> type("Kia ora te ao!")
<type 'str'>
>>> type(17)
<type 'int'>
```

The name of the function is `type`, and it displays the type of a value or variable. The value or variable, which is called the **argument** of the function, has to be enclosed in parentheses. It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.

We can assign the return value to a variable:

```
>>> result = type(17)
>>> print(result)
<type 'int'>
```

Another useful function is **len**. It takes a Python **sequence** as an argument. The only Python sequence we have met so far is a string. A string is a sequence of characters. For a string argument, `len` returns the number of characters the string contains.

```
>>> my_str = "Kia ora te ao"
>>> len(my_str)
11
```

The `len` function can only be used on sequences. Trying to use it on a number, for example, results in an error.

```
>>> len(3)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

## 3.3 More about the print function

The `print` function is different from the `type` function in that it is used to do something (display output) rather than return a result. If we try printing the result returned by a call to `print` we get the following:

[1] There is a full list of built-in functions at https://docs.python.org/3/library/functions.html

```
>>> x = print("Kia ora")
Kia ora
>>> print(x)
None
```

`None` is a special value returned by functions that are more concerned with doing stuff than calculating or determining values. We will discuss `None` in more depth in chapter 6.

When calling `print` we can actually pass in a number of expressions separated by commas. The function will, by default, insert a space between each value.

```
hour = 2
minute = 45
print("Number of minutes since midnight: ", hour * 60 + minute)
```

produces:

```
Number of minutes since midnight:  165
```

This default space can be changed by using the *optional argument* `sep`:

```
print("You", "are", "kidding", sep = "!")
```

which will produce:

```
You!are!kidding
```

By default the print function appends a newline character to the string being printed. We can change this using the optional argument `end`.

```
print("You have got to be kidding", end = "!")
print("Why is this sentence jammed up on the same line?")
```

This gives us:

```
You have got to be kidding!Why is this sentence jammed up on the same line?
```

## 3.4 Function definitions and use

As well as the built-in functions provided by Python you can define your own functions. In the context of programming, a **function** is a named sequence of statements that performs a desired operation. This operation is specified in a **function definition**. In Python, the syntax for a function definition is:

```
def NAME( LIST OF PARAMETERS ):
    STATEMENTS
```

The 'list of parameters' is where the arguments supplied to the function end up. You will see more of this later.

You can make up any names you want for the functions you create, except that you must obey the same rules discussed in Section 2.3. The list of parameters specifies what information, if any, you have to provide in order to use the new function.

There can be any number of statements inside the function, but they have to be indented from the `def`. In the examples in this book, we will use the standard indentation of four spaces.[2] Thonny's tab is set up to be equivalent to 4 spaces and it does automatically indent statements that follow a colon. Function definitions are the first of several **compound statements** we will see, all of which have the same pattern:

1. A **header**, which begins with a keyword and ends with a colon.
2. A **body** consisting of one or more Python statements, each indented the same amount – *4 spaces is the Python standard* – from the header.

In a function definition, the keyword in the header is `def`, which is followed by the name of the function and a list of *parameters* enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters. In either case, the parentheses are required. The first couple of functions we are going to write have no parameters, so the syntax looks like this:

```
def new_line():
    print()
```

This function is named `new_line`. The empty parentheses indicate that it has no parameters (that is it takes no arguments). Its body contains only a single statement, which outputs a newline character. (That's what happens when you call the print function without any arguments.)

Defining a new function does not make the function run. To do that we need a **function call**. Function calls contain the name of the function to be executed followed by a list of values, called *arguments*, which are assigned to the parameters in the function definition. Our first examples have an empty parameter list, so the function calls do not take any arguments. Notice, however, that the *parentheses are required in the function call* :

```
print("First Line.")
new_line()
print("Second Line.")
```

The output of this program is:

```

First line.

Second line.
```

The extra space between the two lines is a result of the `new_line()` function call. What if we wanted more space between the lines? We could call the same function repeatedly:

```
print("First Line.")
new_line()
new_line()
new_line()
print("Second Line.")
```

[2]The Python language definition does not require 4 spaces, but it does require consistency. You could use any number of spaces, but each block must use the same indentation scheme throughout.

Or we could write a new function named three_lines that prints three new lines:

```
def three_lines():
    new_line()
    new_line()
    new_line()

print("First Line.")
three_lines()
print("Second Line.")
```

This function contains three statements, all of which are indented by four spaces. Since the next statement is not indented, Python knows that it is not part of the function. You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one function call another function; in this case three_lines calls new_line.

So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:

1. Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.
2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call three_lines three times.

Pulling together the code fragments from the previous section into a script named functions.py, the whole program looks like this:

```
def new_line():
    print()

def three_lines():
    new_line()
    new_line()
    new_line()

print("First Line.")
three_lines()
print("Second Line.")
```

This program contains two function definitions: new_line and three_lines. Function definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the function is called, and the function definition generates no output. As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

## 3.5 Flow of execution

In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**. Execution always begins at the first statement

of the program. Statements are executed one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called. Although it is not common, you can define one function inside another. In this case, the inner definition isn't executed until the outer function is called.

Function calls are like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off. That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to execute the statements in another function. But while executing that new function, the program might have to execute yet another function! Fortunately, Python is adept at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

It is usually not helpful to read a program from top to bottom. In python programs the top part of a file is almost always used for function definitions and it is not necessary to read those until you want to understand what a particular function does. The bottom part of a python file is often called the **main** program. This part can be recognised because it is often not indented. It is easier to understand a program by following the flow of execution starting at the beginning of the main program.

## 3.6 Parameters and arguments

Most functions require arguments. Arguments are values that are input to the function and these contain the data that the function works on. For example, if you want to find the absolute value of a number (the distance of a number from zero) you have to indicate what the number is. Python has a built-in function for computing the absolute value:

```
>>> abs(5)
5
>>> abs(-5)
5
```

In this example, the arguments to the `abs` function are `5` and `-5`.

Some functions take more than one argument. For example the built-in function `pow` takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.

```
>>> pow(2, 3)
8
>>> pow(3, 2)
9
```

The first argument is raised to the power of the second argument.

The `round` function, not surprisingly, rounds a number. It takes one or two arguments. The first is the number to be rounded and the second (optional) value is the number of decimal places to round to. If the second number is not supplied it is assumed to be zero and the number is returned as an int. Otherwise the number returned is the same type as the first number passed in. The second argument can be negative.

```
>>> round(1.23456789)
1
>>> round(1.23456789, 0)
1.0
>>> round(1.5)
2
>>> round(1.23456789, 2)
1.23
>>> round(1.23456789, 3)
1.235
>>> round(123456789, 2)
123456789
>>> round(123456789, -2)
123456800
```

Another built-in function that takes more than one argument is `max`.

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3*11, 5**3, 512-9, 1024**0)
503
```

The function `max` can be sent any number of arguments, separated by commas, and will return the maximum value sent. The arguments can be either simple values or expressions. In the last example, `503` is returned, since it is larger than `33`, `125`, and `1`.

The `min` function works in a similar way:

```
>>> min(4, 1, 17, 2, 12)
1
```

Here is an example of a user-defined function that has a parameter:

```
def print_twice(some_variable_name):
    print(some_variable_name, some_variable_name)
```

This function takes a single **argument** and assigns it to the parameter named `some_variable_name`. The value of the parameter (at this point we have no idea what it will be) is printed twice, followed by a newline. The name `some_variable_name` was chosen to suggest that the name you give a parameter is up to you, but in general, you want to choose something more descriptive than `some_variable_name`.

In a function call, the value of the argument is assigned to the corresponding parameter in the function definition. In effect, it is as if `some_variable_name = "Spam"` is executed when `print_twice("Spam")` is called; `some_variable_name = 5` is executed when `print_twice(5)` is called; and `some_variable_name = 3.14159` is executed when `print_twice(3.14159)` is called. Any type of argument that can be printed can be sent to `print_twice`. In the first function call, the argument is a string. In the second, it's an integer. In the third, it's a float.

As with built-in functions, we can use an expression as an argument for `print_twice`:

```
>>> print_twice("Spam"*4)
SpamSpamSpamSpam SpamSpamSpamSpam
```

`"Spam"*4` is first evaluated to '`SpamSpamSpamSpam`', which is then passed as an argument to `print_twice`.

## 3.7 Function composition

Just as with mathematical functions, Python functions can be **composed**, meaning that you use the result of one function as the input to another.

```
>>> print_twice(abs(-7))
7 7
>>> print_twice(max(3, 1, abs(-11), 7))
11 11
```

In the first example, `abs(-7)` evaluates to `7`, which then becomes the argument to `print_twice`. In the second example we have two levels of composition, since `abs(-11)` is first evaluated to `11` before `max(3, 1, 11, 7)` is evaluated to `11` and `print_twice(11)` then displays the result.

We can also use a variable as an argument:

```
>>> saying = "Eric, the half a bee."
>>> print_twice(saying)
Eric, the half a bee. Eric, the half a bee.
```

Notice something very important here. The name of the variable we pass as an argument (`saying`) has nothing to do with the name of the parameter (`some_variable_name`). It doesn't matter what the argument is called; here in `print_twice`, we call everybody `some_variable_name`.

## 3.8 Comments

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they are marked with the # symbol:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
percentage = (minute * 100) / 60     # floating point division - may wish to round
```

Everything from the # to the end of the line is ignored—it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code.

## 3.9 Glossary

**function:** A named sequence of statements that performs some useful operation. Functions may or may not take parameters and may or may not produce a result.

**function definition:** A statement that creates a new function, specifying its name, parameters, and the statements it executes.

**Python sequence:** Several data structures in Python are collectively known as "sequences". They are structures for storing a collection of values in a specific order. For example, a string is a sequence of characters.

**compound statement:** A statement that consists of two parts:

1. header - which begins with a keyword determining the statement type, and ends with a colon.
2. body - containing one or more statements indented the same amount from the header.

The syntax of a compound statement looks like this:

```
keyword expression :
    statement
    statement ...
```

**header:** The first part of a compound statement. Headers begin with a keyword and end with a colon (`:`).

**body:** The second part of a compound statement. The body consists of a sequence of statements all indented the same amount from the beginning of the header. The standard amount of indentation used within the Python community is 4 spaces.

**flow of execution:** The order in which statements are executed during a program run.

**parameter:** A name used inside a function to refer to the value passed as an argument.

**argument:** A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

**abs function:** Returns the absolute value of the argument.

**pow function:** Returns the first argument raised to the power of the second one. `pow(x,y)` is equivalent to `x**y`.

**round function:** Rounds a number and returns the floating point value rounded to n-digits digits after the decimal point. If no second parameter is given it will return the number rounded to the nearest int.

**max function:** Returns the largest argument (if given more than one), or the largest item in a sequence (if given a single sequence).

```
>>> max(3, 8, 2, 1)
8
>>> max("hello")
'o'
```

**composition:** The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

**function composition:** Using the output from one function call as the input to another.

**comment:** Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

**main program:** The part at the bottom of a file that is usually not indented. It is easier to understand a program by following the flow of execution starting from the main program.

## 3.10 Laboratory exercises

1. Using Thonny, create a Python script named `functions.py`. Type the definition of the function `new_line` from the lecture at the top of the file, and then add the definition of the function `three_lines` below it. Add a main routine at the bottom of the script (and not indented) which is a call to `three_lines()`. Don't forget to add the parentheses when you call the function. Run the program.

2. Add a function to the file called `nine_lines` that uses `three_lines` to print nine blank lines. Change the call in the main routine to be a call to `nine_lines()` rather than `three_lines()`.

3. Now add a function named `clear_screen` that prints out exactly forty blank lines. Change the main routine call again to call `clear_screen` to check that it works as expected.

4. What would happen if `nine_lines` was above `three_lines` in the script? Why?

5. Start a new script `more_functions.py` and add a function called `powers(n)` that prints out the first 5 powers of a given number.

```
>>> powers(6)
The first 5 powers of 6 are: 1 6 36 216 1296
>>> powers(10)
The first 5 powers of 10 are: 1 10 100 1000 10000
```

   You can test your script by running it and then calling your functions from the Python interpreter.

   **Show your work to a demonstrator here to make sure you are on the right track.**

6. Add another function to `more_functions.py` called `score_summary` which takes 3 arguments, representing scores from 3 judges. The function should determine and display the bottom, top, and average score. Call this function from the interpreter, sending it different data e.g.

```
>>> score_summary(9.5, 7, 8.5)
Max: 9.5 , Min: 7 , Average: 8.333333333333334
>>> score_summary(9, 8, 7)
Max: 9 , Min: 7 , Average: 8.0
```

7. The results (in the question above) are listed without any way of identifying who they refer to. Make the output more meaningful by adding a name string to the parameter list (and supplying a name argument when you call the function).

```
>>> score_summary("Bruce",9.5, 7, 8.5)
Bruce - Max: 9.5 , Min: 7 , Average: 8.333333333333334
>>> score_summary("Fred", 9, 8, 7)
Fred - Max: 9 , Min: 7 , Average: 8.0
```

8. Write `reject` and `accept` functions which can produce acceptance or rejection form letters. The signature should be in a separate function, used by both the reject and accept functions.

```
>>> reject("Bill")
Dear Bill
I am sorry to inform you that you do not have the job
Yours sincerely
Humphrey Hopalong

>>> reject("Amanda")
Dear Amanda
I am sorry to inform you that you do not have the job
Yours sincerely
Humphrey Hopalong

>>> accept("Vicki")
Dear Vicki
I am pleased to inform you that you have the job
Yours sincerely
Humphrey Hopalong
```

9. Recall that `min()` and `max()` take a 'sequence' as an argument and that strings are 'sequences' of characters. Write a function `show_first` which takes a string argument and prints the minimum value in the string. You can test your script from the interpreter or by calling the function in from the script itself (in the `__main__` section of the script). Try it with "dog", xylophone", "elephant", "Elephant"

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 4

# Functions: part 2

Remember the `print_twice` function from the previous lecture? We are going to use it again, but we will change the parameter name to something a bit more sensible.

```
def print_twice(phrase):
    print(phrase, phrase)
```

## 4.1 Variables and parameters are local

```
def print_joined_twice(part1, part2):
    joined = part1 + part2
    print_twice(joined)
```

This function takes two arguments, concatenates them and stores the result in a local variable `joined`. It then calls `print_twice` with `joined` as the argument. `print_twice` prints the value of the argument, twice. We can call the function with two strings:

```
>>> line1 = "Happy birthday, "
>>> line2 = "to you."
>>> print_joined_twice(line1, line2)
Happy birthday, to you. Happy birthday, to you.
```

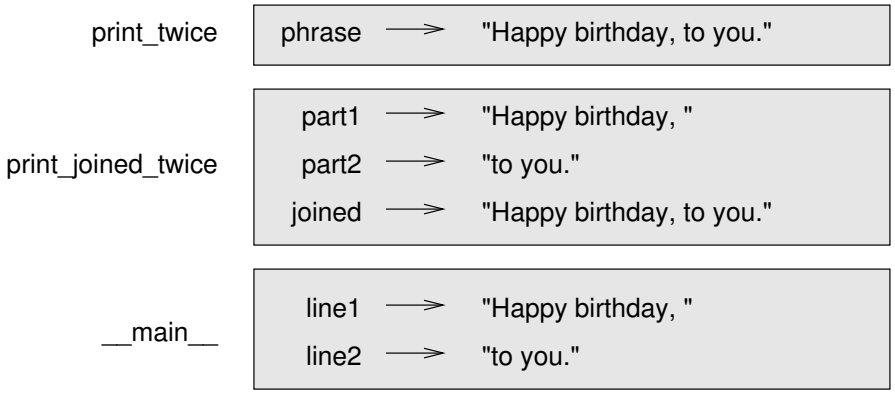
When `print_joined_twice` terminates, the variable `joined` is destroyed. If we try to print it, we get an error:

```
>>> print(joined)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'joined' is not defined
```

When you create a **local variable** inside a function, it only exists inside that function, and you cannot use it outside. Parameters are also local. For example, outside the function `print_twice`, there is no such thing as `phrase`. If you try to use it, Python will complain. Similarly, `part1` and `part2` do not exist outside `print_joined_twice`

## 4.2 Stack diagrams

To keep track of which variables can be used where, it is sometimes useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but they also show the function to which each variable belongs. Each function is represented by a **frame**. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example looks like this:




The order of the stack shows the flow of execution. `print_twice` was called by `print_joined_twice`, and `print_joined_twice` was called by `__main__`, which is a special name for the topmost function. When you create a variable outside of any function, it belongs to `__main__`. Each parameter refers to the same value as its corresponding argument. So, `part1` has the same value as `line1`, `part2` has the same value as `line2`, and `phrase` has the same value as `joined`.

If an error occurs during a function call, Python prints the name of the function, and the name of the function that called it, and the name of the function that called *that*, all the way back to the top most function. To see how this works, we create a Python script named `stacktrace.py` that looks like this:

```
1  def print_twice(phrase):
2      print(phrase, phrase)
3      print(joined)
4
5  def print_joined_twice(part1, part2):
6      joined = part1 + part2
7      print_twice(joined)
8
9  line1 = "Happy birthday, "
10 line2 = "to you."
11 print_joined_twice(line1, line2)
```

We've added the statement, `print(joined)` inside the `print_twice` function, but `joined` is not defined there. Running this script will produce an error message like this:

```
Traceback (most recent call last):
  File "stack-example.py", line 11, in <module>
    print_joined_twice(line1, line2)
  File "stack-example.py", line 7, in print_joined_twice
    print_twice(joined)
  File "stack-example.py", line 3, in print_twice
    print joined
NameError: name 'joined' is not defined
```

This list of functions is called a **traceback**. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error. Notice the similarity between the traceback and the stack diagram. It's not a coincidence. In fact, another common name for a traceback is a *stack trace*.

## 4.3 Input

Remember at the start of the course we said that programs are made up of instructions which only perform a handful of operations[1]. The first two of these operations were input and output. Input involves getting data from the keyboard, a file, or some other device. Output is concerned with displaying data on the screen or sending data to a file or other device.

We have already done quite a bit of displaying data on the screen using the `print` function, which fits into the category of output. We are now going to look at one of the most basic input tasks: getting keyboard input.

There is a built-in function in Python for getting keyboard input, `input`:

```
name = input("Please enter your name: ")
print(name)
num = input("Enter a number: ")
print(num)
```

A sample run of this script would look something like this:

```
>>> %Run tryinput.py
Please enter your name: Arthur, King of the Britons
Arthur, King of the Britons
Enter a number: 435
435
```

This function allows a *prompt* (a message to the user) to be given between the parentheses as an argument to the function. The result returned by `input` is always a string.

```
>>> n = input()
hello
>>> n
'hello'
>>> type(n)
<class 'str'>
>>> n = input()
15
>>> n
'15'
>>> type(n)
<class 'str'>
```

## 4.4 Type conversion

Each Python type comes with a built-in function that attempts to convert values of another type into that type. The `int(ARGUMENT)` function, for example, takes any value and converts it to an integer, if possible,

[1] See the list on page 8 to refresh your memory.

or complains otherwise:

```
>>> int("32")
32
>>> int("Kia ora")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Kia ora'
```

The `int` function can also convert floating-point values to integers, but note that it truncates the fractional part:

```
>>> int(-2.3)
-2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

The `float(ARGUMENT)` function converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

It may seem odd that Python distinguishes the integer value `1` from the floating-point value `1.0`. They may represent the same number, but they belong to different types as they are represented differently inside the computer.

The `str(ARGUMENT)` function converts any argument to type string:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'true' is not defined
```

The `str(ARGUMENT)` function will work with any value and convert it into a string. Note: `True` is a predefined value in Python; true is not. We will see more of `True` in lecture 5.

## 4.5 Glossary

**local variable:** A variable defined inside a function. A local variable can only be used inside its function.

**stack diagram:** A graphical representation of a stack of functions, their variables, and the values to which they refer.

**frame:** A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

**traceback:** A list of the functions that are executing, printed when a runtime error occurs. A traceback is also commonly referred to as a *stack trace*, since it lists the functions in the order in which they are stored in the runtime stack.

## 4.6 Laboratory exercises

1. Create a script named `params.py` with the following contents:

```
def print_param(x):
    print(x)

x = "Kia ora"
print_param("Ka kite")
print(x)
```

   Run the script and record the output of the program.

   Draw a stack diagram showing the value of each function's variables at the moment that the `print` function in `print_param` is called.

2. Create a script called `change_param.py` with the following contents:

```
def change_param(x):
    x = "Ka kite"

x = "Kia ora"
change_param(x)
print(x)
```

   Run the script and record the output of the program in a text file.

   Explain how the output comes about.

3. Try to figure out what the output of the program below will be.

```
def print_all(first, second, third):
    print(first, second, third)

def print_reverse(first, second, third):
    print(third, second, first)

def print_add(first, second, third):
    print(first + second + third)

def print_add_reverse(first, second, third):
    print(third + second + first)

one = "fish"
two = "and"
three = "chips"
print_all(one, two, three)
print_reverse(one, two, three)
print_all(three, two, one)
print_add(one, two, three)
print_add_reverse(one, two, three)
print_all(1, 2, 3)
print_reverse(1, 2, 3)
print_add(1, 2, 3)
print_add_reverse(1, 2, 3)
```

Record your answer, and then type the code into a Python script and run it to check that you are right. Draw the stack diagram at the moment `print_add_reverse(one, two, three)` is called.

4. Type the following code into a file called `input_fun.py`:

```
print("Enter some text:")
text = input()
print("You entered: " + text)
print("Enter some text:")
text = input()
print("You entered: " + text)
print("Enter some text:")
text = input()
print("You entered: " + text)
```

Run the script to see what it does. Simplify the script by identifying the repeated code and placing it in a function. This is called 'encapsulation'.

5. Create a script called `convert.py` which uses the function `input` to read a string from the keyboard. Attempt to convert the string to a float using `float(x)` and also to an integer using `int(x)`. Print out the resulting float and integer. Test the script with the following input:

```
10
103
1e12
0.000001
blah
```

Explain the output produced. Note those that caused errors.

**Show your work to a demonstrator here to make sure you are on the right track.**

6. So far we have used the ',' to print more than one item at a time. In the following exercises this will prove problematic, in particular adding the full stop in the correct place when using a comma always introduces a space by default. Try using '+' to concatenate the various strings instead. Note: concatenation only works with strings so you will need to cast any non-string values.

```
>>> a = "one"
>>> b = "two"
>>> c = 3
>>> print(a, b, c)
one two 3
>>> print(a + b + str(c))
onetwo3
>>> print(a + b + c)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: must be str, not int
```

7. Create a script called `film_critic.py`. In it define a function `rate_movie` that takes an argument `movie` and asks the user to give it a rating out of five. When called like this, for example

```
>>> rate_movie("The Meaning of Life")
```

the function should produce output similar to:

```
Movie: The Meaning of Life; Your rating: 4.
```

8. Define a function that takes two arguments, `name` and `current_age`. When called with the arguments "Eric" and 67, it should produce output similar to:

```
Kia ora Eric, you are 67 years old.
Next year you will be 68.
```

What happens if you pass the parameters in the wrong order?
Try calling the function again, this time passing in values obtained from the user using the `input` function.

## 4.7 Mastery Level 2 - Practice Questions

Mastery Level 2 primarily tests your understanding of material covered in lessons 3 and 4.

There are 2 questions in the test.

You should check your solutions with a demonstrator.

**Part 1 - Scripts with user input**

1. Create a Python script that reads three integers from the user and prints the sum of those numbers. Your program should behave like this (12, 97 and 123 are the numbers entered by the user):

```
 >>>
Enter the first number: 12
Enter the second number: 97
Enter the third number: 123
The sum is: 232
>>>
```

2. Write a Python script that asks the user for first name, last name, institution and position, and then prints the text for a name badge. Your program should behave like this (parts after the colons are entered by the user):

```
>>>
Enter first name: Thomas
Enter last name: Anderson
Enter institution name: MetaCortex
Enter title: Software Developer
Thomas Anderson
Software Developer
MetaCortex
>>>
```

3. Create a Python script that asks the user for a measurement in inches and prints out the measurement in centimeters. Base your calculation on 1 inch being equal to 2.54 cm. Your program should behave like this (where the value 6.5 is entered by the user).

```
>>>
Enter a measurement in inches: 6.5
6.5 inches is equal to 16.51cm.
>>>
```

4. Write a Python script that asks the user how many equally sized pizza slices (sectors) are required from a circular pizza and prints the angle of each slice in degrees. Note: there are 360 degrees in a circle. Your program should behave like this (where the value 12 is entered by the user).

```
>>>
Enter the number of slices you need: 12
Each slice can arc through 30.0 degrees.
>>>
```

**Part 2 - Functions that take arguments**

1. In a script write a function that takes a single argument representing the current minute and prints out the percentage of the hour that has elapsed. When called as follows :

   ```
   percent_hour(30)
   ```

   the function should print:

   ```
   50% of the hour has passed.
   ```

   Add a main routine to your script that calls the function several times, passing in sensible test values of your choosing.

   *Note that there should be no space between the number and the percentage sign.

2. Write a Python script which defines a function called leg_count. The function should take an argument representing an animal's name.

   The function should have a variable called num_legs which gets its input from the keyboard in response to an instruction.

   Make sure you let the user know what is required of them e.g.

   ```
   >>>
   Enter how many legs a crocodile has:
   ```

   Finally the function should finish with a statement which prints out a summary of the information e.g.

   ```
   Apparently a crocodile has 4 legs.
   ```

   Add a main routine that calls the function twice. The first time the argument should be: "crocodile" The second time the argument should be: "trout"

3. Write a function that takes four arguments, a, b, c, d, and prints the remainder when the sum of the first three is divided by d. For example the function might be be called like this:

   ```
   func_name(3,5,7,6)
   ```

   in which case the result printed out would be 3. Add a main routine to your script that calls the function several times, passing in sensible test values of your choosing.

4. In a Python script write a function called book_sticker that takes 2 arguments (title and author) and prints out the line:

   ```
   ON SPECIAL: "title" by "author"!  BUY NOW!
   ```

   Add a main routine that prompts the user for a title and then an author and stores them in two variables. The main routine then calls book_sticker, passing in the user inputs.

   E.g. For the user input "Clean Code" and "Robert C. Martin" it will produce the line:

   ```
   ON SPECIAL: Clean Code by Robert C. Martin!  BUY NOW!
   ```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 5

# Conditionals

## 5.1 Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program accordingly. **Conditional statements** give us this ability. We use this idea of conditional execution very often in our everyday lives. For example, when leaving the house in the morning, you might habitually "execute" the following:

```
if it is windy, take your coat
if it is windy and rainy, take your coat
if it is not windy and it is rainy, take your umbrella.
```

Or, if you live in Dunedin, you probably just take your coat and umbrella every day.

In Python, the simplest form of a conditional statement is the **if statement** :

```
if weather == "windy":
    print("take your coat")
```

The part after the `if` keyword and before the colon is a **boolean expression** and is called the **condition**. A boolean expression is simply an expression that evaluates to `True` or `False`. If the condition is true, then the indented statement gets executed. If the condition is false then the indented statement is skipped and the flow of control jumps to the statement immediately following the `if` block. The syntax for an `if` statement looks like this:

```
if BOOLEAN EXPRESSION:
    STATEMENTS
```

As with the function definition from Lecture 3 and other compound statements, the `if` statement consists of a header and a body. The header begins with the keyword `if` followed by a boolean expression and ends with a colon (`:`). The indented statements that follow are called a **block**. The first unindented statement marks the end of the block. A statement block inside a compound statement is called the **body** of the statement. Each of the statements inside the body are executed in order if the boolean expression evaluates to `True`. The entire block is skipped if the boolean expression evaluates to `False`. There is no limit on the number of statements that can appear in the body of an `if` statement, but there has to be at least one.

## 5.2 Boolean values and expressions

The Python type for storing true and false values is called bool, named after the British mathematician, George Boole. George Boole created *Boolean algebra* , which is the basis of all modern computer arithmetic. There are only two **boolean values** : `True` and `False`. Capitalization is important, since `true` and `false` are not boolean values in Python.

```
>>> type(True)
<class 'bool'>
>>> type(true)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'true' is not defined
```

The operator == compares two values and produces a boolean value:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

In the first statement, the two operands are equal, so the expression evaluates to `True`; in the second statement, 5 is not equal to 6, so we get `False`. The == operator is one of the **comparison operators**; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

## 5.3 Logical operators

Boolean expressions can be combined to create more complicated boolean expressions using one of the three **logical operators** : `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 and x < 10` is true only if `x` is greater than 0 *and* less than 10. `n % 2 == 0 or n % 3 == 0` is true if *either* (or both) of the conditions is true, that is, if the number is divisible by 2 *or* 3. Finally, the `not` operator negates a boolean expression, so `not(x > y)` is true if `(x > y)` is false, that is, if x is less than or equal to y.

Note that the boolean expression `x==5 or 6` is a valid expression but is almost certainly not what you intend. Novice programmers sometimes do the following in an `if` statement:

```
if x==5 or 6:
    print('x is 5 or 6')
```

In fact this boolean expression is always `True`. What is probably intended is:

| x | y | z | Boolean Expression | Value |
|---|---|---|---|---|
| "windy" | "rainy" | - | x=="windy" | |
| "windy" | "rainy" | - | y=="rainy" | |
| "windy" | "rainy" | - | (x=="windy") and (y=="rainy") | |
| "windy" | "sunny" | - | (x=="windy") and (y=="rainy") | |
| "windy" | "sunny" | - | (x=="windy") or (y=="rainy") | |
| True | - | - | x | |
| True | True | - | x or y | |
| True | False | - | x and y | |
| True | False | True | x and (y or z) | |
| 10 | 5 | 7 | (x<y) or (y<z) | |
| 10 | 5 | 7 | (x<y) and (y<z) | |
| 10 | 5 | 7 | (x<y) and (y<x) | |
| 10 | 5 | 7 | (x<y) or (y<z) | |
| 10 | 10 | 7 | (x<y) or (y<z) | |
| 10 | 10 | 7 | ((x<y) or (y<z)) and ((x<20) and (y>5)) | |

Figure 5.1: Test your knowledge of Boolean expressions

```
if x==5 or x==6:
    print('x is 5 or 6')
```

Although the former `if` statement sounds natural to a human, it is almost always the wrong thing to do. Make sure you are explicit with all the subparts of your Boolean expressions.

## 5.4 Alternative execution

A second form of the `if` statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

```
if x % 2 == 0:
    print(x, "is even")
else:
    print(x, "is odd")
```

If the remainder when `x` is divided by `2` is `0`, then we know that `x` is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches** , because they are branches in the flow of execution.

As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might wrap this code in a function:

```
def print_parity(x):
    if x % 2 == 0:
        print(x, "is even")
    else:
        print(x, "is odd")
```

For any value of x, `print_parity` displays an appropriate message. When you call it, you can provide any integer expression as an argument.

```
>>> print_parity(17)
17 is odd.
>>> y = 41
>>> print_parity(y+1)
42 is even.
```

## 5.5 Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
    print(x, "is less than", y)
elif x > y:
    print(x, "is greater than", y)
else:
    print(x, "and", y, "are equal")
```

`elif` is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit to the number of `elif` statements but only a single (and optional) else statement is allowed and it must be the last branch in the statement:

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
elif choice == 'c':
    function_c()
else:
    print("Invalid choice.")
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

## 5.6 Nested conditionals

One conditional can also be **nested** within another. Consider the following example:

```
if x == y:
    print(x, "and", y, "are equal")
else:
    if x < y:
        print(x, "is less than", y)
    else:
        print(x, "is greater than", y)
```

The outer conditional contains two branches. The first branch contains a simple output statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as well. Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. A nested conditional can always be rewritten as a chained conditional and vice versa. What to use is partly a matter of choice, but in general you should use the structure that is easiest to understand.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
    if x < 10:
        print("x is a positive single digit.")
```

The call to the print function is executed only if we make it past both the conditionals, so we can use the *and* operator:

```
if 0 < x and x < 10:
    print("x is a positive single digit.")
```

These kinds of conditions are common, so Python provides an alternative syntax that is similar to mathematical notation:

```
if 0 < x < 10:
    print("x is a positive single digit.")
```

This condition is semantically the same as the compound boolean expression and the nested conditional.

## 5.7 Booleans and type conversion

Python assigns boolean values to values of other types. For numerical types like integers and floating-points, zero values are false and non-zero values are true. For strings, empty strings are false and non-empty strings are true.

```
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'true' is not defined
>>> bool(1)
True
>>> bool(0)
False
>>> bool("Banana")
True
>>> bool("")
False
>>> bool(3.14159)
True
>>> bool(0.0)
False
>>> bool("False")       # "False" is a non-empty string, cf. bool("Banana")
True
```

This can cause real confusion in conditional statements. For example:

```
badWeather = "False"
if badWeather:
    print("Take your coat")
```

will always print `"Take your coat"` (perfect for Dunedin perhaps).

## 5.8 Glossary

**boolean value:** There are exactly two boolean values: True and False. Boolean values result when a boolean expression is evaluated by the Python interpreter. They have type bool.

**boolean expression:** An expression that is either true or false.

**bool function:** Converts the argument to a boolean.

**comparison operator:** One of the operators that compares two values: ==, !=, >, <, >=, and <=.

**logical operator:** One of the operators that combines boolean expressions: and, or, and not.

**conditional statement:** A statement that controls the flow of execution depending on some condition. In Python the keywords if, elif, and else are used for conditional statements.

**condition:** The boolean expression in a conditional statement that determines which branch is executed.

**block:** A group of consecutive statements with the same indentation.

**body:** The block of statements in a compound statement that follows the header.

**branch:** One of the possible paths of the flow of execution determined by conditional execution.

**chained conditional:** A conditional branch with more than two possible flows of execution. In Python chained conditionals are written with `if ... elif ... else` statements.

**nesting:** One program structure within another, such as a conditional statement inside a branch of another conditional statement.

## 5.9 Laboratory exercises

1. Look at the following expressions. What do you think the results would be? Enter the expressions in to the Python interpreter to check your predictions.

```
True or False
True and False
not(False) and True
not(False or True)
not(False and True)
False or not(True)
True and True
True or True
False or False
```

   Analyze these results. In words, explain the behaviour of the `and` and `or` operators.

2.

```
if x < y:
    print(x, "is less than", y)
elif x > y:
    print(x, "is greater than", y)
else:
    print(x, "and", y, "are equal")
```

   Encapsulate (wrap) this code in a function called `compare(x, y)`. Call `compare` three times: one each where the first argument is less than, greater than, and equal to the second argument and check that it prints the appropriate message.

3. A script contains the following code:

```
secret = 55
guess = int(input("Guess a number between 50 and 100: "))
guess_right(secret, guess)
```

   Write the function named `guess_right`. Use `if`, `else` and `!=` to print out a message reporting whether or not the secret number was guessed correctly. Your output for a correct and an incorrect guess could look like this:

```
Guess a number between 50 and 100: 55
You win. 55 is equal to 55
```

```
Guess a number between 50 and 100: 55
You lose. 54 is not equal to 55
```

4. Write a function named `is_divisible_by_3` that takes a single integer as an argument and prints "This number is divisible by three." if the argument is evenly divisible by 3 and "This number is not divisible by three." otherwise. *Hint:* remember the mod operator `%` returns 0 if the first argument is divisible by the second (so `6%3` is 0).

   Now write a similar function named `is_divisible_by_5`.

5. Write a function called `is_divisible` that takes two arguments. The function should print whether the first is divisible by the second. Example outputs might be:

```
>>> is_divisible(3, 2)
3 is not divisible by 2
>>> is_divisible(6, 3)
6 is divisible by 3
```

**Show your work to a demonstrator here to make sure you are on the right track.**

6. Write a function `water_state` which takes one parameter that represents a temperature in degrees centigrade. If the temperature is greater than 100, print "Steam", less than 0, print"Ice", otherwise print "Liquid".
7. Write a function which asks a yes/no question of the user. If the user types any variation of Y, y, Yes, yes, or YES print "You chose Yes" otherwise print "You chose No"
8. Write a script that allows users to select one of three functions to run by doing the following:
   (a) Make a new, empty script called choice.py.
   (b) Add three functions (`function_a`, `function_b` and `function_c`) to the file. When called each function should just print out a message saying it was called. For example:

```
def function_a():
    print("function_a was called")
```

   (c) Add the following code in a function called `dispatch(choice)`:

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
elif choice == 'c':
    function_c()
else:
    print("Invalid choice.")
```

   (d) Run your script and try calling it from the interpreter with different arguments. For example:

```
>>> dispatch('a')
```

   (e) Add some code to the main routine of the script that gets the input from the user and then calls dispatch.
9. Write a function which takes 3 integer arguments and prints "equal" if any of the arguments is equal to any other argument.

```
>>> any_equal(1, 2, 3)
No inputs match.
>>> any_equal(1, 2, 1)
At least two inputs are the same.
>>> any_equal(1, 1, 3)
At least two inputs are the same.
>>> any_equal("ant", "ant", 3)
At least two inputs are the same.
>>> any_equal("ant", "ant", "ant")
At least two inputs are the same.
```

10. Write a function `next_level` which takes 6 scores as arguments. A student needs to have a total score of 200 from the 5 best subjects in order to move on. Write a body for the function which discards the lowest score and totals the rest, printing "Pass" or "Repeat" as appropriate. Hint: remember the `min()` function.

11. *Extension Exercise:* Write a function that takes 3 parameters and prints them out in order. For example:

```
>>> print_in_order(5, 6, 4)
In order: 4, 5, 6
```

12. *Extension Exercise:* Write a function `find_middle` that takes 3 parameters and prints the middle one. For example:

```
>>> print_middle(5, 6, 4)
The middle value is 5
```

13. *Extension Exercise:* To better understand boolean expressions, it is helpful to construct truth tables. Two boolean expressions are *logically equivalent* if they have the same truth table. The following Python function `show_truth_table` prints out the truth table for any boolean expression in two variables `p` and `q`:

    `boolean_ex.py`:

```
def show_boolean_expression(p, q, expression):
    print(str(p) + "\t" + str(q) + "\t" + str(eval(expression)))

def show_truth_table(expression):
    print(" p       q      " + expression)
    length = len(" p       q      " + expression)
    print(length * "-")

    show_boolean_expression(True, True, expression)
    show_boolean_expression(True, False, expression)
    show_boolean_expression(False, True, expression)
    show_boolean_expression(False, False, expression)
```

    There is a new function in the above code: `eval` *evaluates* its argument as a python expression. You can use `eval` to evaluate arbitrary strings. Let's find out more about boolean expressions. Copy this program to a file named `boolean_ex.py`. Run `boolean_ex.py` from IDLE, then use the interpreter window to call `show_truth_table` with various strings. For example:

```
>>> show_truth_table("p or q")
 p       q      p or q
----------------------
True    True    True
True    False   True
False   True    True
False   False   False
>>>
```

    Use the `show_truth_table` function with the following boolean expressions, recording the truth table produced each time:

    (a) not(p or q)

(b) p and q

(c) not(p and q)

(d) not(p) or not(q)

(e) not(p) and not(q)

Which of these are logically equivalent?

14. *Extension Exercise:* This really is quite an advanced question. Type in and call the following function. Then try to come up with a sensible description of what is going on.

```
def adams():
   answer = 42
   guess = int(input("What is the answer to the ultimate question? "))
   if guess < answer:
       print("No silly, that's too small.")
       adams()
   elif guess > answer:
       print("You dope, that's too big.")
       adams()
   else:
       print("Well done. Wonder what the question is?")
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 6

# Functions that return a value

## 6.1 Return values

The built-in functions we have used, such as `abs`, `pow`, `round` and `max`, have produced results. Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression.

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

So far, none of the functions we have written has *returned* a value, they have printed values. In this lecture, we are going to write functions that return values, which we will call **fruitful functions**, for want of a better name. The first example is `area_of_circle`, which returns the area of a circle with the given radius:

```
def area_of_circle(radius):
    if radius < 0:
        print("Warning: radius must be non-negative")
        return

    area = 3.14159 * radius**2
    return area
```

There are two return statements in the above code. In the first, we have a lonely `return` statement which instructs the Python interpreter to return from the function immediately. It is useful to note that *all* functions return a value. If a function has a lonely `return` statement (no value), or if there is no `return` statement, then that function returns the value `None`.

The second `return` statement is accompanied by a return **value**. This statement means: Return immediately from this function and use the following expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area_of_circle(radius):
    if radius < 0:
        print("Warning: radius must be non-negative")
        return

    return 3.14159 * radius**2
```

On the other hand, **temporary variables** like area often make debugging easier. Sometimes it is useful to

have multiple `return` statements, one in each branch of a conditional. For instance, we have already seen the built-in `abs`, now we see how to write our own:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these `return` statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Another way to write the above function is to leave out the `else` and just follow the `if` condition by the second `return` statement.

```
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

Think about this version and convince yourself it works the same as the first one. Code that appears any place the flow of execution can never reach, is called **dead code**. In a fruitful function, it is a good idea to ensure that every possible path through the program hits a `return` statement. The following version of `absolute_value` fails to do this:

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

This version is not correct because if `x` happens to be `0`, neither condition is true, and the function ends without hitting a `return` statement. In this case, the return value is a special value called **None**:

```
>>> print(absolute_value(0))
None
```

`None` is the unique value of a type called the `NoneType`:

```
>>> type(None)
<class 'NoneType'>
```

All Python functions return `None` whenever they do not return another value. So the earlier functions we wrote that didn't have a `return` statement were actually returning values, we just never checked.

```
 >>> def print_hello():
...     print("hello")
...
>>> return_value = print_hello()
hello
>>> type(return_value)
<class 'NoneType'>
```

## 6.2 Program development

At this point, you should be able to look at complete functions and tell what they do. Also, while completing the laboratory exercises given so far, you will have written some small functions. As you write larger functions, you might start to have more difficulty, especially with runtime and semantic errors. To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates $(x_1, y_1)$ and $(x_2, y_2)$. By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)? In this case, the two points are the inputs, which we can represent using four parameters. The return value is the distance, which is a floating-point value. Already we can write an outline of the function:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version of the function doesn't compute distances; it always returns zero. But it is syntactically correct, and it will run, which means that we can test it before we make it more complicated.

To test the new function, we call it with sample values:

```
>>> distance(1, 2, 4, 6)
0.0
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer. At this point we have confirmed that the function is syntactically correct, and we can start adding lines of code. After each incremental change, we test the function again. If an error occurs at any point, we know where it must be—in the last line we added.

A logical first step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We will store those values in temporary variables named `dx` and `dy` and print them.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print("dx is", dx)
    print("dy is", dy)
    return 0.0
```

If the function is working, the outputs should be 3 and 4. If so, we know that the function is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print("dsquared is: ", dsquared)
```

```
    return 0.0
```

Notice that we removed the `print` calls we wrote in the previous step. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product. Again, we would run the program at this stage and check the output (which should be 25). Finally, using the fractional exponent 0.5 to find the square root, we compute and return the result:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of result before the return statement. When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, the incremental development process can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to hold intermediate values so you can output and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

## 6.3 Composition

As you should expect by now, you can call one function from within another. This ability is called **composition**. As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle. Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a function, distance, that does just that, so now all we have to do is use it:

```
radius = distance(xc, yc, xp, yp)
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
result = area_of_circle(radius)
```

Wrapping that up in a function, we get:

```
def area_of_circle_two_points(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area_of_circle(radius)
    return result
```

We called this function area_of_circle_two_points to distinguish it from the area_of_circle function defined earlier. There can only be one function with a given name within a given module. The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def area_of_circle_two_points(xc, yc, xp, yp):
    return area_of_circle(distance(xc, yc, xp, yp))
```

## 6.4 Boolean functions

Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

The name of this function is is_divisible. It is common to give **boolean functions** names that sound like yes/no questions. is_divisible returns either True or False to indicate whether the x is or is not divisible by y. We can make the function more concise by taking advantage of the fact that the condition of the if statement is itself a boolean expression. We can return it directly, avoiding the if statement altogether:

```
def is_divisible(x, y):
    return x % y == 0
```

This session shows the new function in action:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
    print("x is divisible by y")
else:
    print("x is not divisible by y")
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
    print("x is divisible by y")
else:
    print("x is not divisible by y")
```

But the extra comparison is unnecessary.

## 6.5 Glossary

**fruitful function:** A function that yields a return value.

**return value:** The value provided as the result of a function call.

**temporary variable:** A variable used to store an intermediate value in a complex calculation.

**dead code:** Part of a program that can never be executed, often because it appears after a return statement.

**`None`:** A special Python value returned by functions that have no return statement, or a return statement without an argument. `None` is the only value of the type `NoneType`.

**incremental development:** A program development plan intended to simplify debugging by adding and testing only a small amount of code at a time.

**scaffolding:** Code that is used during program development but is not part of the final version.

**boolean function:** A function that returns a boolean value.

**composition (of functions):** Calling one function from within the body of another, or using the return value of one function as an argument to the call of another.

## 6.6 Laboratory exercises

1. Write a function called `is_even(n)` that takes an integer as an argument and returns `True` if the argument is an **even number** and `False` if it is **odd** .
2. Now write the function `is_odd(n)` that returns `True` when `n` is odd and `False` otherwise. Finally, modify `is_odd` so that it uses a call to `is_even` to determine if its argument is an odd integer.
3. Write a function called `is_factor` which returns `True` if the first argument is a factor of the second. That is, `is_factor(3, 12)` is `True` whereas `is_factor(5, 12)` is `False`.
4. Write a function called `hypotenuse` that returns the length of the hypotenuse of a right angled triangle given the length of the other two sides as parameters. Here's an initial definition:

```
def hypotenuse(a,b):
    return 0.0
```

**Show your work to a demonstrator here to make sure you are on the right track.**

5. Copy the `area_of_circle` function definition from Section 6.1. Now write a function called `area_of_snowman` that takes as arguments three numbers representing the radii of the head, middle and bottom sections of a snowman. Return the area of the snowman using calls to `area_of_circle` to calculate the result.
6. Make a copy of your 3 judges program from exercise 6 on page 40. Adapt the function so it *returns* the average score. Adapt the function calls so each value is stored in a variable. Write a statement in the main routine which reports the highest average score.
7. Make a copy of your `next_level` function from exercise 10 on page 62. Adapt the function so it *returns* the best 5 total. Adapt the function calls so each value is stored in a variable. Write a statement in the main routine which reports the highest best 5 score.
8. Make a copy of your `water_state` function from the exercise 6 on page 61. Adapt the function so it *returns* the descriptive string. The main routine should collect the temperature from the user, then display the state e.g. H2O at 300 degrees will be steam.
9. Python functions return `None` whenever they do not return anything else. Using the two functions below show that this statement is true.

```
def func1():
    print("Function 1")
    return

def func2():
    print("Function 2")
```

10. If you didn't complete all the levels of the game in the previous lab try to finish it this time.
11. Write a function that presents a menu as follows:

```
Calculate the area of which shape?
  1 a triangle
  2 a circle
  3 a square
Enter a number to choose:
```

Depending on the user's choice, ask for the appropriate inputs, call an appropriate function to get the correct area and display the result sensibly.

12. Open `film_critic.py` that you wrote in Exercise 7, page 49. Add a function called `get_stars` that takes a number and returns either "BOMB" ( if the argument is zero), a string of up to four stars (depending on the argument), "MUST SEE" (if the argument is 5), or "?". Call this function inside `rate_movie` instead of just printing the numerical rating.

13. *Extension Exercise:* Make a copy of your `find_middle` program from exercise 12 on page 62. Adapt the function so it *returns* the middle value. The main routine should collect 3 integers from the user then display the middle number.

14. *Extension Exercise:* Recall that the equation for a line can be written as $y = mx+c$, where $m$ is the slope of the line and $c$ is the $y$-intercept. For a line that passes through two points, $(x_1, y_1)$ and $(x_2, y_2)$, we have the following identities:

$$\begin{aligned} m &= \frac{y_2 - y_1}{x_2 - x_1} \\ c &= y_1 - mx_1 \end{aligned}$$

    (a) Write a function `slope(x1, y1, x2, y2)` that returns the slope of the line through the points `(x1, y1)` and `(x2, y2)`.

    (b) Write a function `intercept(x1, y1, x2, y2)` that returns the $y$-intercept of the line through the points `(x1, y1)` and `(x2, y2)`.

## 6.7 Mastery Level 3 - Practice Questions

Mastery Level 3 primarily tests your understanding of material covered in lessons 5 and 6.

There are 2 questions in the test.

You should check your solutions with a demonstrator.

**Part 1 - Functions with Boolean parameters and/or a Boolean return type**

1. Write a function that takes a Boolean and prints it out followed by the results of converting it to a string, an int and a float (each on a separate line). Call the function at least twice in the main routine using sensible test values as arguments.
2. Write a function that takes two ints and returns True if the second int is a multiple of the first and False if it is not. Print the results of calling the function at least three times in the main routine using sensible test values as arguments.
3. Write a function that takes 2 arguments and returns True if they are both of the same type and False if they are not. Print the results of calling the function at least three times in the main routine using sensible test values as arguments.
4. In a Python script write a function called boolean_test that takes two Boolean arguments (x and y) and prints out the following Boolean expressions and their respective values:

   x or y

   not (x and y)

   So for example the function could be called like this:

   ```
   boolean_test(True, False)
   ```

   in which case the output should be:

   ```
   True or False is True
   not (True and False) is True
   ```

   Call the function in main several times with a comprehensive selection of test values.

**Part 2 - Functions with if/elif/else and a returned value**

1. In a Python script write a function that returns either "North Island", "South Island" or "unknown" for a city that is passed in as an argument.

   The function should know where Dunedin, Christchurch, Wellington and Auckland are.

   The function should also recognise the city names even if they do not start with a capital letter.

   Add a main routine that asks the user for a city, passes it to the function and prints out the returned value as shown below (where Hamilton is the value entered by the user):

   ```
   >>>
   Enter the city you wish to locate: Hamilton
   The location of Hamilton is unknown
   >>>
   ```

   Add at least three further statements in main to test your function with a good selection of hardcoded values.

2. In a Python script write a function that takes a single integer argument indicating the age of a person and returns the name of the discount package to which they are entitled.

   If the age is between 0 and 14 inclusive, then return

   ```
   "Kids4Free"
   ```

   If the age is greater than 64, then return

   ```
   "GoldenAge"
   ```

   If the age is greater than 14, but less than or equal to 64, then return:

   ```
   "NoDiscountForYou"
   ```

   Add a main routine that asks the user for an age, passes it to the function and prints out the returned value in a sentence as shown below (where 78 is the value entered by the user):

```
>>>
Enter your age: 78
You are entitled to a GoldenAge discount package.
>>>
```

   Add at least three further statements to main to test your function with a good selection of hardcoded values.

   You may assume that the user will enter a non-negative number.

3. The movie ratings of the International Movie Database (IMDb.com) could be described as follows:

   ```
   0 to less than 5:  shockingly bad

   5.0 - less than 7:  strictly for a rainy afternoon

   7.0 - less than 8:  pretty good

   8.0 +:  a must see
   ```

   In a Python script write a function that takes a single argument indicating the rating for a specific movie and returns the appropriate description.

   Add a main routine that asks the user for a rating, passes it to the function and prints out the returned value in a sentence as shown (where 3.3 is the value entered by the user).

```
>>>
Enter the movie rating: 3.3
A rating of 3.3 indicates that this movie is shockingly bad.
>>>
```

   Add at least four further statements to main to test your function with a good selection of hardcoded values as arguments. You may assume that the user will enter a non-negative number.

4. Consider cardinal and ordinal numbers:

| Cardinal | | Ordinal |
|---|---|---|
| 1 | — | 1st |
| 2 | — | 2nd |
| 3 | — | 3rd |
| 4 | — | 4th |
| 5 | — | 5th |
| 11 | — | 11th |
| 111 | — | 111th |
| 121 | — | 121st |
| 112 | — | 112th |
| 222 | — | 222nd etc. |

Write a function that takes an int and returns that number as an ordinal number string.

Numbers ending in 11, 12 and 13 will need to have 'th' appended.

Otherwise:

Numbers ending in 1 will need to have 'st' appended
Numbers ending in 2 will need to have 'nd' appended
Numbers ending with a 3 will need to have 'rd' appended
All other numbers will need 'th' appended.

Add a main routine that asks the user for a cardinal number, passes it to the function and prints out the returned value as shown (where 3 is the value entered by the user).

```
>>> Enter integer: 3
3rd
>>>
```

Add at least three further statements to main to test your function with a good selection of hardcoded values. You may assume that the user will indeed enter a cardinal number.

Hint: Use the modulus operator. The remainder after dividing an int by 10 will be the last digit. The remainder after dividing an int by 100 will be the last two digits.

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 7

# Modules and TDD

This chapter primarily deals with a development methodology called Test Driven Development (TDD). Before we get to TDD though, we need a couple of extra parts of python: modules and triple quoted strings.

## 7.1 Modules, files and the `import` statement

A **module** is simply a Python file that contains definitions (usually function definitions). As programs get large, it makes sense to separate the program into different modules. This allows the programmer to minimize the complexity of the programming task. Typically, similar functionality or related functions are grouped into a single module and separate modules are as independent of each other as possible. Creating your own modules is as easy as creating a Python script with definitions in it; in fact you have already written lots of Python modules.

The Python standard library contains a very large number of modules for many different tasks. It is worthwhile browsing the library just to get a feel for all the useful things it can do. The current documentation can be accessed at: `https://docs.python.org/3/library/index.html`. We will look at the `doctest` module later in this lecture, and throughout the course several other modules will be introduced.

There are three ways of using `import` to import a module and subsequently use what was imported. The `math` module contains many useful math functions, we will use it to illustrate:

1.
```
import math
print(math.cos(math.pi/2.0))
```

   This statement imports every function from the `math` module which are then accessed using dot notation.

2.
```
from math import cos, pi
print(cos(pi/2.0))
```

   This statement imports only the definitions of `cos` and `pi` from the math library. Nothing else is imported.

3.
```
from math import *
print(cos(pi/2.0))
```

This statement also imports everything from the `math` module, the difference being that dot notation is not needed to access module members.

It is more common to use the first or second alternative than the last. The first has the advantage of avoiding naming conflicts (two different modules defining `cos` for example), at the cost of lengthier function calls. The second has the opposite advantages and disadvantages.

We can use the `help` function to see the functions and data contained within modules. The keyword module contains a single function, `iskeyword`, which as its name suggests is a boolean function that returns `True` if a string passed to it is a keyword:

```
>>> from keyword import *
>>> iskeyword("for")
True
>>> iskeyword("all")
False
>>>
```

The data item, `kwlist` contains a list of all the current keywords in Python:

```
>>> from keyword import *
>>> print(kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with', 'yield']
>>>
```

We encourage you to check out https://docs.python.org/3/library to explore the extensive libraries that come with Python. There are so many treasures to discover!

## 7.2 Triple quoted strings

In addition to the single and double quoted strings we first saw in Lecture 2, Python also has *triple quoted strings*. We will need triple quoted strings for Unit Testing (next section). Here are some examples:

```
>>> type("""This is a triple quoted string using 3 double quotes.""")
<class 'str'>
>>> type('''This triple quoted strings uses 3 single quotes.''')
<class 'str'>
>>>
```

Triple quoted strings can contain both single and double quotes inside them:

```
>>> print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')
"Oh no", she exclaimed, "Ben's bike is broken!"
>>>
```

Finally, triple quoted strings can span multiple lines:

```
>>> message = """This message will
... span several
... lines."""
>>> print(message)
This message will
span several
lines.
>>>
```

## 7.3 Unit testing with `doctest`

It is a common best practice in software development these days to include automatic **unit testing** of source code. Unit testing provides a way to automatically verify that individual pieces of code, such as functions, are working correctly. This makes it possible to change the implementation of a function at a later time and quickly test that it still does what it was intended to do. Python has a built-in doctest module for easy unit testing. Doctests can be written within a triple quoted string on the *first line* of the body of a function or script. They consist of sample interpreter sessions with a series of inputs to a Python prompt followed by the expected output from the Python interpreter. The doctest module automatically runs any statement beginning with >>> (followed by a space) and compares the following line with the output from the interpreter. To see how this works, put the following in a script named `first_doctest.py`[1]:

```
def is_divisible_by_2_or_5(n):
    """
    >>> is_divisible_by_2_or_5(8)
    True
    """


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

The last three lines are what make doctest run. Put them in the main routine of any file that includes doctests. Running the script will produce the following output:

```
**********************************************************************
File "first_doctest.py", line 3, in __main__.is_divisible_by_2_or_5
Failed example:
    is_divisible_by_2_or_5(8)
Expected:
    True
Got nothing
**********************************************************************
1 items had failures:
   1 of   1 in __main__.is_divisible_by_2_or_5
***Test Failed*** 1 failures.
```

This is an example of a *failing test*. The test says: if you call `is_divisible_by_2_or_5(8)` the result should be `True`. Since `is_divisible_by_2_or_5` as written doesn't return anything at all, the test fails, and doctest tells us that it expected `True` but got nothing.

[1] There are four underscores in `__name__`, two at the start and two at the end.

We can make this test pass by returning True:

```
def is_divisible_by_2_or_5(n):
    """
    >>> is_divisible_by_2_or_5(8)
    True
    """
    return True


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If we run it now, there will be no output, which indicates that the test passed. Note again that the doctest string must be placed immediately after the function definition header in order to run. To see more detailed output, add the keyword argument `verbose=True` to the `testmod` method call

```
doctest.testmod(verbose=True)
```

and run the module again. This will produce output showing the result of running the tests and whether they pass or not.

```
Trying:
    is_divisible_by_2_or_5(8)
Expecting:
    True
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.is_divisible_by_2_or_5
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

While the test passed, our test suite is clearly inadequate, since `is_divisible_by_2_or_5` will now return `True` no matter what argument is passed to it. Here is a completed version with a more complete test suite and code that makes the tests pass:

```
def is_divisible_by_2_or_5(n):
    """
    >>> is_divisible_by_2_or_5(8)
    True
    >>> is_divisible_by_2_or_5(7)
    False
    >>> is_divisible_by_2_or_5(5)
    True
    >>> is_divisible_by_2_or_5(9)
    False
    """
    return n % 2 == 0 or n % 5 == 0


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Run the module again and see what you get.

## 7.4 Good Practice Development

The difficult part of programming, ironically, is not the programming part. The difficult part is the problem solving part. That is, deciding how to solve the problem at hand. Most programming problems can be solved by splitting the problem into smaller problems until you get to problems you can solve. Then you join up those smaller solutions to solve the larger problem. Of course, it is not always obvious to know how to split up a large problem and this is where experience and practice plays a key role. Nevertheless, here are some things you can try when presented with a problem you have no idea how to solve.

Try splitting the problem into two sub-problems, and progressively split each sub-problem until you know how to solve something. For example:

1. Count the number of "the"s in a string.
   (a) split the string into separate words
   (b) count the number of words that are equal to "the"
      i. test if a single word equals "the"
      ii. scan a list of words and keep count of those that equal "the"

Even if you don't know how to solve a given sub-problem, and you can't think of how to split it further, at least you have a better chance of discovering how to solve a simpler problem (either by asking someone, or asking Google for example).

There are also simple rules to follow when you are developing your own test cases. You should include:

- typical input
- atypical input (e.g. input at the extreme edge of allowable input)
- incorrect or invalid input
- the simplest type of input (if that makes sense)
- the most complicated type of input (if that makes sense)

- enough tests so that each line of code is exercised at least once in your test set (this is called test coverage).

## 7.5 Programming with style

Readability is very important to programmers, since in practice programs are read and modified far more often then they are written. All the code examples in this book will be consistent with the *Python Enhancement Proposal 8* (PEP 8), a style guide developed by the Python community. We'll have more to say about style as our programs become more complex, but a few pointers will be helpful already:

- use 4 spaces for indentation
- imports should go at the top of the file
- separate function definitions with two blank lines
- keep function definitions together
- keep top level statements, including function calls, together in the main routine of the program
- use Test-driven development to develop your programs (this is not part of PEP 8, but is more a philosophy of software development).

## 7.6 Glossary

**unit testing:** An automatic procedure used to validate that individual units of code are working properly. Python has doctest built in for this purpose.

**module:** A file containing Python definitions and statements intended for use in other Python programs. The contents of a module are made available to the other program by using the import statement.

**standard library:** A library is a collection of software used as tools in the development of other software. The standard library of a programming language is the set of such tools that are distributed with the core programming language. Python comes with an extensive standard library.

**import statement:** A statement which makes the objects contained in a module available for use. There are three forms for the import statement. Using a hypothetical module named `mymod` containing functions `f1` and `f2`, and variables `v1` and `v2`, examples of these three forms include:

```
import mymod
```

and:

```
from mymod import f1, f2, v1, v2
```

and:

```
from mymod import *
```

**namespace:** A syntactic container providing a context for names so that the same name can reside in different namespaces without ambiguity. In Python, modules, classes, functions and methods all form namespaces.

**naming collision:** A situation in which two or more names in a given namespace cannot be unambiguously resolved. Using

```
import mymodule
```

instead of

```
from mymodule import *
```

prevents naming collisions.

**attribute:** A variable defined inside a module. Module attributes are accessed by using the **dot operator** (.).

**dot operator:** The dot operator (.) permits access to attributes and functions of a module.

**docstring** A string constant on the first line of a function or module definition (and as we will see later, in class and method definitions as well). Docstrings provide a convenient way to associate documentation with code. Docstrings are also used by the doctest module for automated testing.

## 7.7 Laboratory exercises

All of the exercises below should be added to a file named `doctest_ex.py` that contains the following in the main routine:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

After completing each exercise in turn, run the program to confirm that the doctests for your new function pass.

1. Write a compare function that returns `1` if `a > b`, `0` if `a == b`, and `-1` if `a < b`.

   ```
   def compare(a, b):
       """
       Returns 1 if a>b, 0 if a equals b, and -1 if a<b
       >>> compare(5, 4)
       1
       >>> compare(7, 7)
       0
       >>> compare(2, 3)
       -1
       >>> compare(42, 1)
       1
       """
       #  Your function body should begin here.
   ```

   Fill in the body of the function so the doctests pass.

2. Use incremental development to write a function called hypotenuse that returns the length of the hypotenuse of a right triangle given the lengths of the other two sides as parameters. Record each stage of the incremental development process as you go.

   ```
   def hypotenuse(a, b):
       """
       Compute the hypotenuse of a right triangle with sides of length a
       and b.
       >>> hypotenuse(3, 4)
       5.0
       >>> hypotenuse(12, 5)
       13.0
       >>> hypotenuse(7, 24)
       25.0
       >>> hypotenuse(9, 12)
       15.0
       """
   ```

   When you are finished add your completed function with the doctests to `doctest_ex.py` and confirm that the doctests pass.

   **Show your work to a demonstrator here to make sure you are on the right track.**

3. Write a body for the function definition of `fahrenheit_to_celsius` designed to return the integer value of the nearest degree Celsius for a given temperature in Fahrenheit. Use your favourite web

search engine to find the equation for doing the conversion if you don't already know it. (*Hint:* you may want to make use of the built-in function, `round`. Try typing `help(round)` in a Python shell and experimenting with round until you are comfortable with how it works.)

```
def fahrenheit_to_celsius(t):
    """
    >>> fahrenheit_to_celsius(212)
    100
    >>> fahrenheit_to_celsius(32)
    0
    >>> fahrenheit_to_celsius(-40)
    -40
    >>> fahrenheit_to_celsius(36)
    2
    >>> fahrenheit_to_celsius(37)
    3
    >>> fahrenheit_to_celsius(38)
    3
    >>> fahrenheit_to_celsius(39)
    4
    """
```

4. Add a function body for `celsius_to_fahrenheit` to convert from Celsius to Fahrenheit.

```
def celsius_to_fahrenheit(t):
    """
    >>> celsius_to_fahrenheit(0)
    32
    >>> celsius_to_fahrenheit(100)
    212
    >>> celsius_to_fahrenheit(-40)
    -40
    >>> celsius_to_fahrenheit(12)
    54
    >>> celsius_to_fahrenheit(18)
    64
    >>> celsius_to_fahrenheit(-48)
    -54
    """
```

5. *Extension Exercise:* Write a function to convert kilograms to pounds. Devise a sensible test suite for your function.

6. *Extension Exercise:* Write a function that takes the width, height and depth of a cuboid and returns its surface area. It should return -1 if an argument is negative. Devise a sensible test suite for your function.

7. *Extension Exercise:* Write a function that returns the volume of a square based pyramid. It should take as arguments the length of a bottom edge and the height. It should return -1 if either parameter is less than 0. Your function should pass the doctests below.

```
def vol_of_a_pyramid(edge, height):
    """
    >>> vol_of_a_pyramid(0, 3)
    0.0
    >>> vol_of_a_pyramid(3, 0)
    0.0
    >>> vol_of_a_pyramid(2, 3)
    4.0
    >>> vol_of_a_pyramid(4, 3)
    16.0
    >>> vol_of_a_pyramid(6, 4)
    48.0
    >>> vol_of_a_pyramid(5, 4.5)
    37.5
    >>> vol_of_a_pyramid(-2, 4)
    -1
    >>> vol_of_a_pyramid(2, -4)
    -1
    """
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 8

# Strings

## 8.1 A compound data type

So far we have seen a number of types, including: `int`, `float`, `bool`, `NoneType` and `str`. Strings are qualitatively different from the first four because they are made up of smaller pieces—characters. Types that comprise smaller pieces are called **compound data types**. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This flexibility is useful.

The bracket operator selects a single character from a string:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print(letter)
```

The expression `fruit[1]` selects the character from position number 1 in `fruit`. The variable `letter` refers to the result. When we display `letter`, we get a surprise:

```
a
```

The letter at position number 1 in "banana" is actually `a` if you are a computer scientist. For perverse reasons, computer scientists always start counting **index** (or position) numbers from zero. The letter at the *beginning* of "banana" is obviously `b`. To access it, you just put 0, or any expression with the value 0, in the brackets:

```
>>> letter = fruit[0]
>>> print(letter)
b
```

The expression in brackets is called an **index**. An index specifies a member of an ordered list, in this case the list of characters in the string. The index *indicates* which one you want, hence the name. It can be any integer expression. [1]

[1] There are two plural forms of index, **indices** and **indexes**. **Indices** is usually preferred in mathematical or technical contexts.

## 8.2 Length

The `len` function returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
last = fruit[length]       # ERROR!
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no letter at index 6 in "banana". Since we started counting indices at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from length:

```
length = len(fruit)
last = fruit[length-1]
```

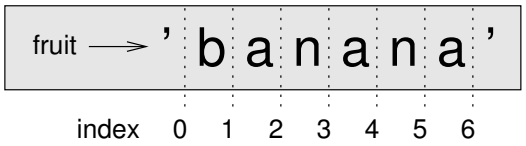
Alternatively, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

## 8.3 String slices

A substring of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = "Peter, Paul, and Mary"
>>> print(s[0:5])
Peter
>>> print(s[7:11])
Paul
>>> print(s[17:21])
Mary
```

The operator `[n:m]` returns the part of the string from the $n^{th}$ character to the $m^{th}$ character, including the first but excluding the last. If you find this behaviour counterintuitive it might make more sense if you imagine the indices pointing *between* the characters, as in the following diagram:




If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

What do you think `s[:]` means?

## 8.4 String comparison

The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
    print("Yes, we have no bananas!")
```

Other comparison operations are useful for putting words in alphabetical order:

```
if word < "banana":
    print("Your word," + word + ", comes before banana.")
elif word > "banana":
    print("Your word," + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")
```

You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

```
Your word, Zebra, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

## 8.5 Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
greeting = "Kia ora te ao!"
greeting[0] = "M"            # ERROR!
print(greeting)
```

Instead of producing the output `Mia ora te ao!`, this code produces the runtime error `TypeError: 'str' object doesn't support item assignment`. Strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Kia ora te ao!"
newGreeting = "M" + greeting[1:]
print(newGreeting)
```

The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

Do you think the following will work?

```
greeting = "Kia ora te ao!"
greeting = "M" + greeting[1:]
print(greeting)
```

## 8.6 Traversal and the `for` loop

Recall (on page 8) we said that all programming languages allowed you to perform a few basic operations: get input, display output, do math, do conditional execution and then there was just one more thing. The last thing we need to add to the list is *repetition*, the ability to *loop* through a set of statements repeatedly.

We will look at this in a lot more detail later (Lectures 10 and 11) but there is a special type of loop that is particularly useful with strings (and other compound types) which is worth introducing while we are looking at strings.

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. Python provides a very useful language feature for traversing many compound types—the `for` loop:

```
for char in fruit:
    print(char)
```

The above piece of code can be understood as an abbreviated version of an English sentence: "For each character in the string fruit, print out the character". The `for` loop is an example of an *iterator*: something that visits or selects every element in a structure (in this case a string), usually in turn. The `for` loop also works on other compound types such as lists and tuples, which we will look at later.

The following example shows how to use concatenation and a `for` loop to generate an abecedarian series. Abecedarian refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for letter in prefixes:
    print(letter + suffix)
```

The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because Ouack and Quack are misspelled. You'll fix this as an exercise below.

## 8.7 The `in` operator

The `in` operator tests if one string is a substring of another:

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

Note that a string is a substring of itself:

```
>>> "a" in "a"
True
>>> "apple" in "apple"
True
```

Combining the `in` operator with string concatenation using `+`, we can write a function that removes all the vowels from a string:

```
def remove_vowels(s):
    # vowels contains all the letters we want to remove
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    # scan through each letter in the input string
    for letter in s:
        # check if the letter is not in the disallowed list of letters
        if letter not in vowels:
            # the letter is allowed, add it to the result
            s_without_vowels += letter
    return s_without_vowels
```

Test this function to confirm that it does what we wanted it to do.

## 8.8 `str` Methods

In addition to the functions that we have seen so far there is also a special type of function called a **method**. You can think of a method as a function which is attached to a certain type of variable (e.g. a string).

When calling a function you just need the name of the function followed by parentheses (possibly with some arguments inside). In contrast a method also needs to be associated with a variable (also called an object). The syntax that is used is the variable (or object) name or a value followed by a dot followed by the name of the method along with possibly some arguments in parentheses like this:

```
VARIABLE.METHODNAME(ARGUMENTS)
```

You can see that it looks just like a function call except for the variable name and the dot at the start. Compare how the `len` function and the `upper` method are used below.

```
>>> my_str = "hello world"
>>> len(my_str)
11
>>> my_str.upper()
'HELLO WORLD'
```

The `len` function returns the length of the sequence which is given as an argument. The `upper` method returns a new string which is the same as the string that it is called upon except that each character has been converted to uppercase. In each case the original string remains unchanged.

An example of a method which needs an argument to operate on is the `count` method.

```
>>> my_str = "the quick brown fox jumps over the lazy dog."
>>> my_str.count("the")
2
>>> my_str.count("hello")
0
>>> my_str.count("e")
3
```

The count method returns the number of times the string given as an argument occurs within the string that it is called upon. But what about the following:

```
>>> ms = "ahaha"
>>> ms.count("aha")
```

The `str` type contains useful methods that manipulate strings. To see what methods are available, use the `dir` function with `str` as an argument.

```
>>> dir(str)
```

which will return the list of items associated with strings:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

To find out more about an item in this list, we can use the `help` command:

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
```

We can call any of these methods using **dot notation**:

```
>>> s = "brendan"
>>> s.capitalize()
'Brendan'
```

`str.find` is a useful method for finding substrings in a string. To find out more about it, we can print out its **docstring**, `__doc__`, which contains documentation on the function:

```
>>> print(str.find.__doc__)
S.find(sub [,start [,end]]) -> int

Return the lowest index in S where substring sub is found,
such that sub is contained within S[start,end].  Optional
arguments start and end are interpreted as in slice notation.

Return -1 on failure.
```

Calling the `help` function also prints out the docstring:

```
>>> help(str.find)
Help on method_descriptor:

find(...)
    S.find(sub [,start [,end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start,end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

The parameters in square brackets are optional parameters. We can use `str.find` to find the location of a character in a string:

```
>>> fruit = "banana"
>>> index = fruit.find("a")
>>> print(index)
1
```

Or a substring:

```
>>> fruit.find("na")
2
```

It also takes an additional argument that specifies the index at which it should start:

```
>>> fruit.find("na", 3)
4
```

And has a second optional parameter specifying the index at which the search should end:

```
>>> "bob".find("b", 1, 2)
-1
```

In this example, the search fails because the letter *b* does not appear in the index range from 1 to 2 (not including 2).

## 8.9 The split method and lists

Quite often, we are interested in the words that are contained within a string. The `split` method in `str` allows us to break a string up into a **list** of words. We will look at lists in more depth in Lesson 12. Lists and strings are similar but while a string is specifically a sequence of characters, the elements of a list can be of any type. These elements are separated by commas and enclosed in square brackets:

```
>>> my_list = [23, "peaches", 4.9, False]
```

As with strings, you can access a single element using square brackets containing the appropriate index number:

```
>>> first_item = my_list[0]
>>> print(first_item)
23
```

Slices also work in the same way:

```
>>> print(my_list[1:3])
['peaches', 4.9]
```

We can use the len function to determine the length of a list:

```
>>> print(len(my_list))
4
```

The result of calling the split method on a string is a list of strings:

```
>>> s = "The quick brown fox jumps over the lazy dog."
>>> words  = s.split()
>>> print(words)
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog.']
```

By default the string is split on white space. What do you think s.split('e') would produce?

Just as we used the `for` loop to traverse a string one character at a time, we can now use it to traverse our list one word at a time. The following function finds the length of the longest word in a string:

```
def find_longest_word_length(s):

    words = s.split()
    longest = 0
    for word in words:
        if len(word) > longest:
            longest = len(word)
    return longest
```

Test this function to confirm that it does what we intend it to do.

The in operator also works as we might expect:

```
>>> words = ["qwerty",  "type", "giraffe", "manic"]
>>> "qwerty" in words
True
>>> "asdfgh" in words
False
```

If I want to check that a string contains a certain word e.g. 'the', why should I split the string first?

## 8.10 Problem Solving Patterns

In this chapter and some of the following chapters, we're going to look at common patterns that can be used to solve certain classes of problems. These patterns are meant to be general but easily adapted to the specific problem. Building up your vocabulary of patterns is an important aspect of becoming a good programmer. Each of the following subsections is an example of a particular string pattern.

### 8.10.1 Print characters in a string

```
def print_a_string_pattern(s):
    for ch in s:
        # possible if statement here
        print(ch)
```

For example, to print only the vowels in a given string:

```
def print_vowels(s):
    for ch in s:
        if ch in 'aeiouAEIOU':
            print(ch)
```

### 8.10.2 Build a new string

```
def build_a_string_copy(s):
    result = ''    # initialise the resulting string to empty
    for ch in s:
        # possible if statement here
        result += ch   # add the new character to the end of the result string
    return result
```

For example, to replace vowels with 'X":

```
def replace_vowels_with_X(s):
    result = ''    # initialise the resulting string to empty
    for ch in s:
        if ch in 'aeiouAEIOU':
            result += 'X'   # add an X instead of ch
        else:
            result += ch    # add ch
    return result
```

### 8.10.3 Count characters

```
def count_characters(s):
    count = 0     # initialise the number of characters we are counting to 0
    for ch in s:
        # possible if statement here
        count += 1   # count the current character by incrementing our count variable
    return count
```

For example, to count the number of times the letter `a` appears in a string:

```
def count_a_s(s):
    count = 0  # no 'a' has been found yet
    for ch in s:
        if ch == 'a':
            count += 1 # the character that we are up to is an 'a' so increment count
    return count
```

## 8.11 Glossary

**compound data type:** A data type in which the values are made up of components, or elements, that are themselves values.

**index:** A variable or value used to select a member of an ordered set, such as a character from a string.

**traverse:** To iterate through the elements of a set, performing a similar operation on each.

**slice:** A part of a string (substring) specified by a range of indices. More generally, a subsequence of any sequence type in Python can be created using the slice operator (*sequence* [*start*:*stop*]).

**immutable:** A compound data types whose elements can not be assigned new values.

**method:** A function that is attached to a particular type and can be accessed via dot notation. For example ``monday''.upper()

## 8.12 Laboratory exercises

1. Replace the '?' to achieve the shown output.

```
>>> saying = "Be as happy as possible"
>>> saying[?:?]
'Be'
>>> saying[?:?]
'happy'
>>> saying[?:?]
'Be as happy as possible'
>>> print( ? ) # use slices and concatenation
'Be as happy as humanly possible'
>>> print( ? ) # use concatenation and len() function
The number of letters in saying is: 23
```

2. Write a function `on_many_lines` that takes a string as an argument and uses a `for` loop to print the letter one per line. For example:

```
>>> on_many_lines("banana")
b
a
n
a
n
a
```

3. Record the output after executing the following statements:

```
>>> saying = "Be as happy as possible"
>>> saying.swapcase()

>>> saying.upper()

>>> saying.count("e")

>>> saying[0].isupper()

>>> saying[:2].isalpha()
```

**Show your work to a demonstrator here to make sure you are on the right track.**

4. Modify:

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for letter in prefixes:
    print(letter + suffix)
```

so that Ouack and Quack are spelled correctly. You will need to use an `if` statement inside the `for` loop to decide if a different suffix is needed.

5. Encapsulate (or wrap):

```
fruit = "banana"
count = 0
for char in fruit:
    if char == "a":
        count += 1
print(count)
```

in a function named `count_letters`, and generalise it so that it accepts the string and the letter as arguments. When you are finished you will be able to call the function like this: `count_letters("banana", "a")` to count the 'a's in 'banana' or like this: `count_letters("fruit fly", "f")` to count the 'f's in 'fruit fly' and so on.

6. Write a function `only_upper` that takes a string as an argument and uses a `for` loop to print only the uppercase letters. For example:

```
>>> saying = "Be As Happy As Possible"
>>> only_upper(saying)
BAHAP
```

You will need to use the `isupper` method. See `help(str.isupper)` for more information.

7. Write a function that takes a string converts it to a binary string where vowels are replaced by 0 and consonants are replaced by 1. The function should return the binary string. Here are the doctests:

```
def binary_string(s):
    """
    >>> binary_string("robot")
    '10101'
    >>> binary_string("Kia ora te ao!")
    '100 010 10 00!'
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

8. Write a function that takes a string and returns the total scrabble value of the letters. You may assume the rack only contains uppercase letters or an underscore (which represents a blank and is worth zero). Remember the extremely useful `in` operator:

**1 point** E, A, I, O, N, R, T, L, S, U

**2 points** D, G

**3 points** B, C, M, P

**4 points** F, H, V, W, Y

**5 points** K

**8 points** J, X

**10 points** Q, Z

```
def scrabble_score(rack):
    """
    >>> scrabble_score("AABBWOL")
    14
    >>> scrabble_score("QWERTYU")
    22
    >>> scrabble_score("QUIZ_ED")
    25
    """

    if __name__ == '__main__':
        import doctest
        doctest.testmod(verbose = True)
```

9. Write a function that takes a string, s, and an int, x. It should return a new string where every xth character (starting at zero) is now followed by an asterisk.

```
def string_chunks(string, x):
    """
    >>> string_chunks("Once upon a time, in a land far, far away", 5)
    'O*nce u*pon a* time*, in *a lan*d far*, far* away*'
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

10. Create a file named `stringtools.py` and put the following in it:

```
def remove_letter(letter, strng):
    """
    >>> remove_letter("a", "apple")
    'pple'
    >>> remove_letter("a", "banana")
    'bnn'
    >>> remove_letter("z", "banana")
    'banana'
    >>> remove_letter("i", "Mississippi")
    'Mssssppi'
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Write a function body for it that will make it work as indicated by the doctests. Remember that the strings (such as 'pple') should be *return*ed by the function and not printed by the function (and this is the case for almost all doctest tested functions).

11. Include `reverse` in `stringtools.py`.

```
def reverse(s):
    """
    >>> reverse("happy")
    'yppah'
    >>> reverse("Python")
    'nohtyP'
    >>> reverse("")
    ''
    >>> reverse("P")
    'P'
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Add a function body to `reverse` to make the doctests pass. There are two ways to do this exercise. One way uses a for loop over the input string and adds each character to the *start* of the newly constructed string. The second way uses slicing.

12. Add `mirror` to `stringtools.py`.

```
def mirror(s):
    """
    >>> mirror("good")
    'gooddoog'
    >>> mirror("yes")
    'yessey'
    >>> mirror("Python")
    'PythonnohtyP'
    >>> mirror("")
    ''
    >>> mirror("a")
    'aa'
    """
```

Write a function body for it that will make it work as indicated by the doctests. Hint: use `reverse` from the previous exercise to create a second string.

13. *Extension Exercise:* Write a function `remove_duplicates` which takes a string argument and returns a string which is the same as the argument except only the first occurrence of each letter is present. Make your function case sensitive, and ensure that it passes the doctests below.

```
def remove_duplicates(strng):
    """
    >>> remove_duplicates("apple")
    'aple'
    >>> remove_duplicates("Mississippi")
    'Misp'
    >>> remove_duplicates("The quick brown fox jumps over the lazy dog")
    'The quick brown fx jmps v t lazy dg'
    """
```

14. *Extension Exercise:* Convert a date read from the user, given in DD/MM/YYYY format into written format. For example, here's a run of how the program might work:

```
Enter a date in DD/MM/YY format: 16/7/2003
Output: 16th July, 2003
```

There are two problems to solve here: converting a numeric month into the name for the month, and getting the correct suffix to the day (st, nd, rd, th). Both problems can be solved with judicious use of lists.

## 8.13 Mastery Level 4 - Practice Questions

Mastery Level 4 primarily tests your understanding of material covered in lessons 7 and 8.

There are 2 questions in the test.

You should check your solutions with a demonstrator.

**Part 1 - String slices and methods**

1. Write a function that takes a string, s, and a single character, ch, and returns a slice of s up to and including the first occurrence of ch. If the character is not present it should return an empty string. The function should pass the following doctests:

```
def find_slice(s, ch):
    '''
    >>> find_slice("abcdefghijk", "f")
    'abcdef'
    >>> find_slice("aaabbbccc", "b")
    'aaab'
    >>> find_slice("aaabbbccc", "d")
    ''
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

2. Write a function that takes a string and returns the first half of it. Exclude the middle character if the string is uneven in length. Remember that slice indices must be integers. It should pass the following doctests:

```
def first_half(s):
    '''
    >>> first_half("abcdef")
    'abc'
    >>> first_half("abcdefg")
    'abc'
    >>> first_half("a")
    ''
    >>> first_half("good dogs")
    'good'
    '''
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

3. Write a function that takes a string and returns the second half of it. Exclude the middle character if the string is uneven in length. Remember that slice indices must be integers. It should pass the following doctests:

```
def second_half(s):
    '''
    >>> second_half("abcdef")
    'def'
    >>> second_half("abcdefg")
    'efg'
    >>> second_half("a")
    ''
    >>> second_half("good dogs")
    'dogs'
    '''
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

4. File extensions

   Write a function that takes a string parameter representing a file name complete with file extension and returns just the name of the file with no extension.

   Print out the result of calling your function in the main routine.

   HINT: You will need to find the highest index of '.'. Check out `help(str.rfind)`

   The function should pass the following doctests.

```
def extension_remover(file_name):
    '''
    >>> extension_remover("my_text_file.txt")
    'my_text_file'
    >>> extension_remover("data.2015.docx")
    'data.2015'
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

5. Write a function called multi_blank that accepts two parameters, a string, s, and an integer, n. The function must use slice notation to return a string consisting of the first n original letters and the remaining letters replaced by '*'.

   The doctests below should make this clear:

```
def multi_blank(strng, ch_count):
    '''
    >>> multi_blank("banana", 1)
    'b*****'
    >>> multi_blank("bread", 1)
    'b****'
    >>> multi_blank("banana", 2)
    'ba****'
    >>> multi_blank("banana", 3)
    'ban***'
    >>> multi_blank("chips", 0)
    '*****'
    '''

if __name__== "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

**Part 2 - String methods and for loops**

1. Write a function that takes a string as an argument and returns the number of vowels in the sentence. Remember the reserved word `in`.
   The function should pass the following doctests.

```
def count_vowels(sentence):
    """
    Return the number of vowels (a,e,i,o,u) in a sentence.
    >>> count_vowels("Kia ora te ao!")
    7
    >>> count_vowels("baa baa do baa baa")
    9
    >>> count_vowels("HELLO")
    2
    >>> count_vowels("zzz")
    0
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

2. Write a function that takes a string as an argument and returns the number of upper case letters in the sentence. Remember the str method `isupper()`

   The function should pass the following doctests.

```
def count_upper(sentence):
    """
    >>> count_upper("Kia ora te ao!")
    1
    >>> count_upper("27th of April 1968")
    1
    >>> count_upper("HELLO")
    5
    >>> count_upper("zzz")
    0
    """
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

3. Match Brackets

   Write a function that takes a string and checks that it contains matching parentheses i.e (). Return True if it does and False if it doesn't.

   Hint: Consider using a variable to keep count of unclosed parentheses: +1 for an opening one and -1 for a closing one. Use this to check you haven't closed one before you opened one and that all of them have been closed at the end.

   The function should pass the following doctests.

```
def match_brackets(s):
    '''
    >>> match_brackets('(7 - 4) * (3 + 2)')
    True
    >>> match_brackets('((2 + 5) / (13 +12)')
    False
    >>> match_brackets(')14 + 12) % 3')
    False
    >>> match_brackets(')(')
    False
    >>> match_brackets('())(')
    False
    >>> match_brackets('())(()()')
    False
    >>> match_brackets('())()(')
    False
    '''

if __name__== "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

4. Ord numbers

   Remember: Each Unicode character has a corresponding integer. This int can be found using the ord function e.g.

```
ord('a') # returns 97
ord('b') # returns 98
ord('z') # returns 122
ord('A') # returns 65
ord('Z') # returns 90
```

   Write a function that takes a string and loops through it. It should return a string containing the ord numbers of each character in the original, separated by spaces.
   The function should pass the following doctests.

```
def get_ords(s):
    '''
    >>> get_ords('abc')
    '97 98 99 '
    >>> get_ords('a b c')
    '97 32 98 32 99 '
    >>> get_ords('a1 b2 c3')
    '97 49 32 98 50 32 99 51 '
    >>> get_ords('[(!)]')
    '91 40 33 41 93 '
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

5. Balloon Strings

   Remember: Each Unicode character has a corresponding integer. This int can be found using the ord function e.g.

```
    ord('a') # returns 97
    ord('b') # returns 98
    ord('z') # returns 122
    ord('A') # returns 65
    ord('Z') # returns 90
```

   Write a function called balloon_string that takes a string, s, and an int, x. The function should return a new string derived from the argument string but with some characters repeated and some removed. To achieve this each character is multiplied by its ord mod x.

   E.g. Given the string 'abc' and the int 7, 'a' will occur 6 times since 97 % 7 = 6, b will occur 0 times since 98 % 7 = 0 and c will occur just once since 99 % 7 = 1.

The function should pass the following doctests.

```
def balloon_string(string, x):
    '''
    >>> balloon_string('abcdef', 4)
    'abbccceff'
    >>> balloon_string('A great day!', 4)
    'Agggrreaay!'
    >>> balloon_string('ABC 1234', 3)
    'AAC  1224'
    '''

if __name__== "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 9

# Files

Remember at the start of the course we said that programs are made up of instructions which only perform a handful of operations[1]. The first two of these operations were input and output. Input involves getting data from the keyboard, a file, or some other device. Output is concerned with displaying data on the screen or sending data to a file or other device.

We have already seen how to generate output to the screen and get input from the keyboard. In this lecture we are going to look at getting input from a file and generating output to a file.

## 9.1 Files

While a program is running, its data is stored in *random access memory* (**RAM**). RAM is fast and inexpensive, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time you turn on your computer and start your program, you have to write it to a **non-volatile** storage medium, such as a hard drive, USB drive, or optical disk. Data on non-volatile storage media is stored in named locations called **files**. By reading and writing files, programs can save information between program runs.

Working with files is a lot like working with a notebook. To use a notebook, you have to open it. When you're done, you have to close it. While the notebook is open, you can either write in it or read from it. In either case, you know where you are in the notebook. You can read the whole notebook in its natural order or you can skip around. All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.

Opening a file creates a file object. In this example, the variable `myfile` refers to the new file object.

```
>>> myfile = open("test.txt", "w")
>>> print(myfile)
<_io.TextIOWrapper name='test.txt' mode='w' encoding='US-ASCII'>
```

The `open` function takes two arguments. The first is the name of the file, and the second is the **mode**. Mode `"w"` means that we are opening the file for writing. If there is no file named `test.txt`, it will be created. If there already is one, it will be replaced by the file we are writing.

When we print the file object we see the type of the file, name of the file, the mode, and the encoding scheme. The type is TextIOWrapper - a file object that deals with text and the only kind we will deal with in this course.

[1] See the list back on page 8 to refresh your memory.

To put data in the file we invoke the `write` method on the file object. As well as writing data to the file it returns the number of characters written (the length of the string):

```
>>> myfile.write("Now is the time")
15
>>> myfile.write("to close the file")
17
```

Closing the file tells the system that we are done writing and makes sure that everything actually gets written to disk.

```
>>> myfile.close()
```

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is `"r"` for reading:

```
>>> myfile = open("test.txt", "r")
```

Actually the second argument is optional, since the default mode when opening a file is `"r"`. So we could have just done:

```
>>> myfile = open("test.txt")
```

If we try to open a file for reading that doesn't exist, we get an error:

```
>>> myfile = open("toast.txt")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'toast.txt'
```

Not surprisingly, the `read` method of a file reads data from the file. With no arguments, it reads the entire contents of the file into a single string:

```
>>> text = myfile.read()
>>> print(text)
Now is the timeto close the file
```

There is no space between `time` and `to` because we did not write a space between the strings.

The `read` method can also take an argument that indicates how many characters to read:

```
>>> myfile = open("test.txt")
>>> print(myfile.read(5))
Now i
```

If not enough characters are left in the file, `read` returns the remaining characters. When we get to the end of the file, `read` returns the empty string:

```
>>> print(myfile.read(1000006))
s the timeto close the file
>>> print(myfile.read())

>>>
```

## 9.2 Processing things from a file

You would normally read information from a file in a Python program because you want to do something with it. Let's try counting the number of words in a file. Here is a short file for us to work with.

```
Mary had a little lamb
little lamb little lamb
Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
Mary went Mary went
everywhere that Mary went
the lamb was sure to go
```

We can start by opening the file and reading its contents into a string.

```
>>> myfile = open("mary.txt")
>>> text_string = myfile.read()
>>> text_string
'Mary had a little lamb\nlittle lamb little lamb\nMary had a little
lamb\nits fleece was white as snow\nand everywhere that Mary
went\nMary went Mary went\neverywhere that Mary went\nthe lamb was
sure to go\n'
```

Those funny little `\n` characters which you can see in `text_string` represent newlines in the file. They illustrate the very important distinction between **representation** and **presentation**. Pretty much all computer programs try to separate out these concepts. Clearly, it would be difficult to use the same representation and presentation for a newline since newlines are not visible in the usual sense. Rather, they are examples of a **control character**, which goes back to the days of teletypes and typewriters, that instructs the device to do something other than print. In the case of newlines, the instruction is to go to the next line before resuming printing. Python still needs ways of representing these control characters, and it adopts the same notation as many other programming languages where control characters are specified by backslashes. If you want to explicitly see how a string is represented in Python, you can use the function `repr`:

```
>>> a = 'hello\nthere'
>>> a
'hello\nthere'
>>> print(a)
hello
there
>>> print(repr(a))
'hello\nthere'
```

Now that we have the entire file in `text_string` we can use `split` to get a **list** of words from the string. A `list` is another python sequence type and has many properties similar to a string. A list is designated by comma separated values of any type surrounded by square braces. We will see more of lists in Lecture 12, but for now just think of them as a sequence of values. Once we have a list, we can use the `len` function to tell us how many items are in the list. The number of items in the list will be exactly the same as the number of words in the file.

```
>>> word_list = text_string.split()
>>> word_list
['Mary', 'had', 'a', 'little', 'lamb', 'little', 'lamb', 'little',
'lamb', 'Mary', 'had', 'a', 'little', 'lamb', 'its', 'fleece', 'was',
'white', 'as', 'snow', 'and', 'everywhere', 'that', 'Mary', 'went',
'Mary', 'went', 'Mary', 'went', 'everywhere', 'that', 'Mary', 'went',
'the', 'lamb', 'was', 'sure', 'to', 'go']
>>> len(word_list)
39
```

What if we wanted to create a new file which contained a poem about John instead of Mary? We already have the poem about Mary in `text_string` so all we need to do is change every occurrence of Mary to John and then write the new poem to a file.

```
>>> new_poem = text_string.replace("Mary", "John")
>>> new_poem
'John had a little lamb\nlittle lamb little lamb\nJohn had a little
lamb\nits fleece was white as snow\nand everywhere that John
went\nJohn went John went\neverywhere that John went\nthe lamb was
sure to go\n'
>>> myfile = open("john.txt", "w")
>>> myfile.write(new_poem)
198
>>> myfile.close()
```

## 9.3 Directories

Files on non-volatile storage media are organized by a set of rules known as a **file system**. File systems are made up of files and **directories**, which are containers for both files and other directories. When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.

If you open a python interpreter with Thonny, the current working directory may not be where you think. To find out what directory we're currently in, we need to use the `os` module which holds information about various aspects of the operating system. Here is what happens on my Linux machine:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\rcrimp'
```

`C:\\Users\\rcrimp` is Reuben's home directory and happens to be where I started Thonny from.

If you want to open a file somewhere else, you have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

```
>>> wordsfile = open("C:\\Users\\rcrimp\\Documents\\COMP151\\Lab9\\words.txt", "r")
>>> wordlist = wordsfile.readlines()
>>> print(wordlist[:5])
['A\n', 'a\n', 'aa\n', 'aal\n', 'aalii\n']
```

This example opens a file named `words` that resides in a directory named `dict`, which resides in `share`, which resides in `usr`, which resides in the top-level directory of the system, called `/`. It then reads in each

line into a list using `readlines`, and prints out the first 5 elements from that list. You cannot use / as part of a filename; it is reserved as a **delimiter** between directory and filenames. The file `/usr/share/dict/words` should exist on unix based systems, and contains a list of words in alphabetical order.

Alternatively, you can change the current working directory using the method `os.chdir(DIRECTORYNAME)`, where `DIRECTORYNAME` is a string indicating the new directory. For example, to change the current working directory **to** your home directory, try the following:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\rcrimp\\'
```

The function `os.path.expanduser("~")` returns a string representing your home directory.

## 9.4 Reading csv files

For scientists, a fairly common operation involves working and analysing data. That data often comes in the form of a comma separated values (**csv**) file. Most scientific type languages (matlab, R, Julia) have mechanisms for reading and writing csv files that simplifies the process, and Python is no exception. In fact, there are several ways to do it in Python, but here we'll look at just the simplest. This involves using the `csv` module. Here I'm going to use a simple example of reading in a csv file that contains temperature data since 1950 for a temperature station near Omarama. Here is the code:

```
import csv

csvfile = open('Tara_Hills_temp.csv')
csvreader = csv.reader(csvfile)
for row in csvreader:
    print(row)
```

This code uses a csvreader which is a special object that satisfies the **iterator** protocol and behaves in a similar way to how we used an iterator for strings in Lesson 8. The first few lines of output are given below:

```
['YEAR', 'JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG', 'SEP', 'OCT', 'NOV', 'DEC']
['1949', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', 'NA', '12.17', '12.70', 'NA']
['1950', '15.28', '14.84', '12.66', '8.63', '6.44', '2.28', '2.71', '3.03', '7.64', '11.09', '12.88', '13.91', '14.27']
['1951', '15.38', '15.04', '13.57', '8.73', '3.62', '0.05', '1.37', '2.22', '7.63', '9.26', '11.20', '12.35', '14.78']
['1952', '13.66', '15.22', '13.11', '9.21', '5.64', '1.73', '1.00', '4.48', '7.81', '10.74', '11.18', '15.22', '13.74']
['1953', '15.22', '14.73', '13.15', '9.74', '5.63', '1.61', '1.53', '4.30', '6.96', '9.20', '12.78', '14.43', '15.06']
['1954', '16.34', '16.64', '13.61', '8.51', '6.38', '4.43', '1.56', '3.35', '7.13', '9.82', '14.15', '15.08', '15.80']
```

## 9.5 Processing data from a csv file

Almost always, when we read in a csv file, we want to do something with the contents. Quite often that something involves some numerical calculations such as computing an average value. However, there is a bit of a problem because all the data in the csv file is represented as a string and we can't compute averages over strings. First, we have to convert the relevant strings to numbers and we can use the `float` function to do the conversion. But there is also a problem with that as the Tara Hills data shows - there are entries where there is no recorded temperature and instead there is a "NA" (meaning not available) in its place. If we try to execute `float('NA')`, then we get an error:

```
>>> float('NA')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'NA'
```

This is an example of a runtime error, or an **exception**. We can deal with exceptions using a `try ... except` block which "catches" generated errors. We can use it in an `is_float` function like so:

```
def is_float(value):
    try:
        fval = float(val)
        return True
    except ValueError:
        return False
```

If the attempt to convert `val` is successful, then `True` is returned, but if it isn't successful a `ValueError` exception is raised which is "caught" by the `except` statement. Without the `try ... except` block, the function would report an error and quit the entire program, but with the block, the error is handled gracefully.

We are going to use `is_float` to compute the average temperature for January. The basic algorithm is:

```
for each row in the csv file:
    get the second element of the list
    add it to a sum of the elements if it's a float, ignore it if not
divide the sum by the number of elements
```

We should probably also ignore the first row, but our `is_float` function handles that nicely, so we can treat it like any other row. The actual code is as follows:

```
import csv

def is_float(value):
    try:
        fval = float(value)
        return True
    except ValueError:
        return False

csvfile = open('../Data/Tara_Hills_temp.csv')

csvreader = csv.reader(csvfile)

sumtemps = 0.0   # initialise sum
num_temps = 0    # initialise the number of items
for row in csvreader:
    val = row[1] # this is January
    if is_float(val):
        sumtemps += float(val)
        num_temps += 1

avg = sumtemps/num_temps
print('Average temperature for January is: ', avg)
```

Which produces the following output:

```
Average temperature for January is:  15.889850746268655
```

## 9.6 Common File Reading Patterns

### 9.6.1 Read a whole file

The simplest pattern is to read the whole file into a string, and then apply whatever string pattern is relevant to the problem. The pattern looks like this:

```
filename = 'words.txt'      # or whatever the filename is
wordfile = open(filename)   # first open the file
text_str = wordfile.read()  # read entire file into string text_str
wordfile.close()            # close the file
# do something with text_str depending on the problem
```

For example, to print the non-vowels in the file:

```
filename = 'words.txt'      # or whatever the filename is
wordfile = open(filename)   # first open the file
text_str = wordfile.read()  # read entire file into string text_str
wordfile.close()            # close the file
for ch in text_str:
    if ch not in 'aeiouAEIOU':
        print(ch, end="")
```

### 9.6.2 Read in one line at a time

This is recommended if you are unsure of the length of the file or if the length is very large.

We discuss it in 10.8.3 since we will need to know how to use a while loop.

### 9.6.3 Read in all lines

Finally, it is also possible to read in all lines in one go, where each line is a separate string and all the strings are put into a list:

```
filename = 'words.txt'      # or whatever the filename is
wordfile = open(filename)   # first open the file
all_lines = wordfile.readlines() # reads in all lines
wordfile.close()            # close the file
for line in all_lines:
    # do something with line
```

To repeat the vowel example above:

```
filename = 'words.txt'      # or whatever the filename is
wordfile = open(filename)   # first open the file
all_lines = wordfile.readlines() # reads in all lines
one_line = wordfile.readline()  # read the next line
wordfile.close()            # close the file
for line in all_lines:
    for ch in line:      # this is an example of a nested loop
        if ch not in 'aeiouAEIOU':
            print(ch, end="")
```

The variable `all_lines` in this example is a list of strings. We haven't officially looked at lists yet. For now you can think of them as behaving a little like a string, except each element in the list can be any value, whereas the elements in a string are characters. Note that the `readlines` method keeps the '\n' character at the end of each line.

## 9.7 Glossary

**volatile memory:** Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

**non-volatile memory:** Memory that can maintain its state without power. Hard drives, flash drives, and optical disks (CD or DVD) are each examples of non-volatile memory.

**file:** A named entity, usually stored on a hard drive, floppy disk, or CD-ROM, that contains a stream of characters.

**mode:** A distinct method of operation within a computer program. Files in Python can be opened in one of three modes: read ("r"), write ("w"), and append ("a").

**directory:** A special type of file that contains other files (and directories).

**path:** A sequence of directory names that specifies the exact location of a file.

**text file:** A file that contains printable characters organized into lines separated by newline characters.

**list:** A sequence of values separated by commas and surrounded by square braces. E.g. `[1, 2, 3]` or `['comma', 'separated', 'values']`.

**control character:** A non-printing character that can be part of a string and is specified by a backslash followed by an actual character. They are instructions for how to print out strings. For example the newline control character `\n` tells Python to go to the next line before continuing to print.

## 9.8 Laboratory exercises

1. Write a program/script that will open a file (you can specify the filename in your code) for reading and print its contents to the screen. For the examples given below we used a file containing:

```
I have a bunch
of red roses
```

2. Write a function which takes a filename as its argument and returns a string with all `\n` characters replaced by the "_" character:

```
>>> one_line("words.txt")
'I have a bunch_of red roses_'
```

### Show your work to a demonstrator here to make sure you are on the right track.

3. Modify the program in Section 9.5 so that it first asks the user for a month, and then prints the average for that month.[2]

4. Modify the program in Section 9.5 so that it prints out the minimum and maximum temperature for a given month.

5. Modify the program in Section 9.5 so that it prints out the average temperature for each year.

6. Write a function which takes 2 arguments, a filename and an integer (called target). The function should return the word at the target position. Doctests:

```
>>> find_word("words.txt", 0)
'I'
>>> find_word("words.txt", 1)
'have'
```

7. Write a function that takes 2 arguments, a filename and a string and returns `True` if the string appears in the file.

8. Write a function which takes 3 arguments, a filename and 2 strings (called target1 and target2). The function should return `True` if the target words are both found in the file.

```
>>> find_words("words.txt", "I", "a")
True
>>> find_words("words.txt", "i", "a")
True
>>> find_words("words.txt", "you", "a")
False
```

9. Extend your program from the first question so that it asks the user for a file to print, and then prints out the contents of that file.

10. Extend your program so that it first prints out all the files in the current working directory before asking for a file to print. *Hint:* use the `os.listdir(".")` function to list the files in the current directory (don't forget to `import os` first).

### Show your work to a demonstrator here to make sure you are on the right track.

11. Modify your program (perhaps create a copy of it), so that it asks the user for two file names and prints out which file is longer (has more characters).

12. Modify your program so that it prints out which of the two files has more words.

[2] You can find the Tara Hills data set at Blackboard → Coursefiles → Data sets.

13. *Extension Exercise:* Extend your answer from Question 10 so that if the requested file is a directory it will list the contents of the directory instead of printing the contents of a file. Make use of the function `os.path.isdir` to tell if a given file is a directory.

14. *Extension Exercise:* Modify your program from Question 5 so that it asks for a month instead of a column number, then computes the average for that particular month.

15. *Extension Exercise:* Write a program that reads in a file and calculates the average word length.

## 9.9 Mastery Level 5 - Practice Questions

Mastery Level 5 primarily tests your mastery of material covered in lesson 9.

There are 2 questions in the test.

You will need to download support files - these are available on Blackboard. Don't alter the files - they are needed for the doctests to work.

You should check your solutions with a demonstrator.

**Part 1 - File reading and processing**

1. Write a function that takes a single string representing a filename as an argument.

   The function should open the file and return the number of letters (not digits or other characters, only upper and lower case letters) that it contains.

   Remember to have all the required files in the same directory as the python file.

   Your function should pass the doctests below:

```
def letter_count(file_name):
    """
    >>> letter_count("ernest_rutherford.txt")
    3245
    >>> letter_count("frances_oldham_kelsey.txt")
    4235
    """
```

2. Write a function called count_possible_sentences that takes the name of a file as an argument and returns how many words end in '!', '?' or '.'.

   Please note that this approach is unlikely to count all sentences present in the text. What might it miss?

   Remember to have all the required files in the same directory as the python file.

   Your function should pass the doctests below:

```
def count_possible_sentences(file_name):
    """
    >>> count_possible_sentences("frances_oldham_kelsey.txt")
    45
    >>> count_possible_sentences("ernest_rutherford.txt")
    32
    >>> count_possible_sentences("marie_curie.txt")
    24
    """
```

3. Write a function called find_last_capitalised that takes a file name as an argument and returns the last word in the file to start with a capital letter. If none of them do, it should return an empty string.

   Remember to have all the required files in the same directory as the python file.

   Note: You do not have to strip punctuation. Commas and full stops etc are just treated as a part of the word.

The function should pass the following doctests:

```
def find_last_capitalised(file_name):
    """
    >>> find_last_capitalised('ernest_rutherford.txt')
    'British'
    >>> find_last_capitalised('tycho_brahe.txt')
    'Together,'
    >>> find_last_capitalised('test_5.txt')
    ''
    """
```

**Part 1 - File reading and writing**

1. Write a function called capitalise_words that takes two file names. The first file name is the file to be read and processed, the second is the destination file for your answer.

   The function should open the given file for processing and then write out to the destination file. In the destination file each word from the given file should be capitalised (i.e. the first letter should be a capital and the rest should be lower case) and followed by a space. You should make use of the capitalize method from str.

   Note: You do not have to strip punctuation. Commas and full stops etc are just treated as a part of the word.

   Remember to have all the required files in the same directory as the python file.

   Your function should pass the doctests below:

```
def capitalise_words(file_name, destination):
    """
    >>> capitalise_words("ernest_rutherford.txt", "rutherford_capitalised.txt")
    >>> f1 = open("rutherford_capitalised.txt")
    >>> text1 = f1.read()
    >>> f2 = open("rutherford_answer_capitalised.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    >>> capitalise_words("marie_curie.txt", "curie_capitalised.txt")
    >>> f1 = open("curie_capitalised.txt")
    >>> text1 = f1.read()
    >>> f2 = open("curie_answer_capitalised.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    """
```

2. Write a function called strip_punctuation that takes two file names. The first file name is the file to be read and processed, the second is the destination file for your answer.

   The function should open the given file for processing and then write out to the destination file. In the destination file any character that is whitespace should be replaced by a single space. Any other character that is not a letter or a digit should be removed.

Your function should pass the doctests below:

```
def strip_punctuation(file_name, destination):
    """
    >>> strip_punctuation("tycho_brahe.txt", "brahe_no_punc.txt")
    >>> f1 = open("brahe_no_punc.txt")
    >>> text1 = f1.read()
    >>> f2 = open("brahe_answer_no_punc.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    >>> strip_punctuation("frances_oldham_kelsey.txt", "kelsey_no_punc.txt")
    >>> f1 = open("kelsey_no_punc.txt")
    >>> text1 = f1.read()
    >>> f2 = open("kelsey_answer_no_punc.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    """
```

3. Write a function called save_word_lengths that takes two file names. The first file name is the file to be read and processed, the second is the destination file for your answer.

   The function should open the given file for processing and then write out to the destination file. In the destination file you should write the length of each word from the source file followed by a space.

   Your function should pass the doctests below:

```
def save_word_lengths(file_name, destination):
    """
    >>> save_word_lengths("tycho_brahe.txt", "brahe_word_lengths.txt")
    >>> f1 = open("brahe_word_lengths.txt")
    >>> text1 = f1.read()
    >>> f2 = open("brahe_answer_word_lengths.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    >>> save_word_lengths("frances_oldham_kelsey.txt", "kelsey_word_lengths.txt")
    >>> f1 = open("kelsey_word_lengths.txt")
    >>> text1 = f1.read()
    >>> f2 = open("kelsey_answer_word_lengths.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    """
```

4. Write a function called speed_reader that takes two file names. The first file name is the file to be read and processed, the second is the destination file for your answer.

   The function should open the given file for processing. It should then write out to the destination file all the words from the first file but with the first three letters in place and the rest of the word replaced with stars. If the word is less than 3 characters long, simply write out the whole word.

   E.g the word umbrella would be written in the answer file as umb*****

   Each word in the destination file should be followed by a space.

   Remember to have all the required files in the same directory as the python file.

Note: You do not have to strip punctuation. Commas and full stops etc are just treated as a part of the word.
Your function should pass the doctests below:

```
def speed_reader(file_name, destination):
    """
    >>> speed_reader("tycho_brahe.txt", "brahe_speed_reader.txt")
    >>> f1 = open("brahe_speed_reader.txt")
    >>> text1 = f1.read()
    >>> f2 = open("brahe_answer_speed_reader.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    >>> speed_reader("frances_oldham_kelsey.txt", "kelsey_speed_reader.txt")
    >>> f1 = open("kelsey_speed_reader.txt")
    >>> text1 = f1.read()
    >>> f2 = open("kelsey_answer_speed_reader.txt")
    >>> text2 = f2.read()
    >>> text1 == text2
    True
    """
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 10

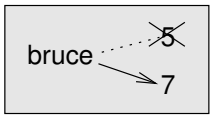
# Iteration: part 1

## 10.1 Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print(bruce, end=' ')
bruce = 7
print(bruce)
```

The output of this program is `5 7`, because the first time `bruce` is printed, his value is `5`, and the second time, his value is `7`. The use of the optional argument `end` with the value `' '` in the first print call replaces the newline (that comes after the output by default) with a space. This is why both outputs appear on the same line. Here is what **multiple assignment** looks like in a state diagram:




With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not! First, equality is symmetric and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not. Furthermore, in mathematics, a statement of equality is always true. If $a = b$ now, then $a$ will always equal $b$. In Python, an assignment statement can make two variables equal, but they don't have to stay that way. Remember we can check equality using double equal signs, ==:

```
a = 5
b = a    # a and b are now equal
print(a == b) # will print True
a = 3    # a and b are no longer equal
print(a == b) # will print False
```

The fourth line changes the value of `a` but does not change the value of `b`, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as <- or :=, to avoid confusion.)

## 10.2 Updating variables

One of the most common forms of multiple assignment is an update, where the new value of the variable depends on the old.

```
x = x + 1
```

This means get the current value of `x`, add one, and then update `x` with the new value. If you try to update a variable that doesn't exist, you get an error, because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left:

```
>>> x = x + 1
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
>>> x
1
>>>
```

Updating a variable by adding `1` is called an **increment**; subtracting `1` is called a **decrement**.

## 10.3 Abbreviated assignment

Incrementing a variable is so common that Python provides an abbreviated syntax for it:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
>>>
```

`count += 1` is an abbreviation for `count = count + 1`. The increment value does not have to be `1`:

```
>>> n = 2
>>> n += 5
>>> n
7
>>>
```

Python also allows the abbreviations `-=`, `*=`, `/=`, `//=`, and `%=`:

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
>>> n
6
>>> n //= 2
>>> n
3
>>> n %= 2
>>> n
1
>>> n = 14
>>> n /= 5
>>> n
2.8
```

## 10.4 The `while` statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Repeated execution of a set of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier. You have already seen one such feature: the `for` loop. A more general feature is the `while` statement.

Here is a function called `countdown` that demonstrates the use of the `while` statement:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n-1
    print("Blastoff!")
```

You can almost read the `while` statement as if it were English. It means, while `n` is greater than `0`, continue displaying the value of `n` and then reducing the value of `n` by `1`. When you get to `0`, display the word Blastoff! More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `False` or `True`.
2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo – Lather, rinse, repeat – are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to `0`. In other cases, it is not so easy to tell:

```
def collatz_sequence(n):
    while n != 1:
        print(n, end = ' ')
        if n % 2 == 0:          # n is even
            n = n // 2
        else:                   # n is odd
            n = n * 3 + 1
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is `1`, which will make the condition false. Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by `2`. If it is odd, the value is replaced by `n * 3 + 1` . For example, if the starting value (the argument passed to `collatz_sequence`) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1. Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach `1`, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches `1`. The previous example ends with such a sequence, starting with `16`. Particular values aside, the interesting question is whether we can prove that this program terminates for *all* values of `n`. So far, no one has been able to prove it *or* disprove it!

## 10.5 Tracing a program

To write effective computer programs a programmer needs to develop the ability to **trace** the execution of a computer program. Tracing involves simulating the computer and following the flow of execution through a sample program run, recording the state of all variables and any output the program generates after each instruction is executed. To understand this process, let's trace the call to `collatz_sequence(3)` from the previous section. At the start of the trace, we have a local variable, `n` (the parameter), with an initial value of `3`. Since `3` is not equal to `1`, the `while` loop body is executed. `3` is printed and `3 % 2 == 0` is evaluated. Since it evaluates to `False`, the else branch is executed and `3 * 3 + 1` is evaluated and assigned to `n`. To keep track of all this as you hand trace a program, make a column heading on a piece of paper for each variable created as the program runs and another one for output. Our trace so far would look something like this:

```
 n              output
---             ------
 3                 3
10
```

Since `10 != 1` evaluates to `True`, the loop body is again executed, and `10` is printed. `10 % 2 == 0` is true, so the `if` branch is executed and `n` becomes `5`. By the end of the trace we have:

```
 n              output
---             ------
 3                3
10               10
 5                5
16               16
 8                8
 4                4
 2                2
 1
```

Tracing can be a bit tedious and error prone (that's why we get computers to do this stuff in the first place!), but it is an essential skill for a programmer to have. From this trace we can learn a lot about the way our code works. We can observe that as soon as `n` becomes a power of `2`, for example, the program will require $\log_2(n)$ executions of the loop body to complete. We can also see that the final `1` will not be printed as output.

## 10.6 Counting digits

The following function counts the number of decimal digits in a positive integer expressed in decimal (i.e. base 10) format:

```
def num_digits(n):
    count = 0
    while n > 0:
        count = count + 1
        n = n // 10
    return count
```

A call to `num_digits(710)` will return 3. Trace the execution of this function call to convince yourself that it works. This function demonstrates another pattern of computation called a **counter** . The variable `count` is initialized to `0` and then incremented each time the loop body is executed. When the loop exits, `count` contains the result – the total number of times the loop body was executed, which is the same as the number of digits. Note the importance of using **int division** in this example! Try it with `n = n / 10` to see. If we wanted to only count digits that are either 0 or 5, adding a conditional before incrementing the counter will do the trick:

```
def num_zero_and_five_digits(n):
    count = 0
    while n > 0:
        digit = n % 10
        if digit == 0 or digit == 5:
            count = count + 1
        n = n // 10
    return count
```

Confirm that `num_zero_and_five_digits(1055030250)` returns 7.

## 10.7 Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make

that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors. When computers appeared on the scene, one of the initial reactions was, "This is great! We can use the computers to generate the tables, so there will be no errors." That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete. Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium used to perform floating-point division.

Although a log table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
x = 1
while x < 13:
    print(x, "\t", 2**x)
    x += 1
```

The string '\t' represents a **tab** character. The backslash character in '\t' indicates the beginning of an **escape sequence** . Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a **newline**. An escape sequence can appear anywhere in a string; in this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a print call, the cursor normally goes to the beginning of the next line. The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program:

```
1       2
2       4
3       8
4       16
5       32
6       64
7       128
8       256
9       512
10      1024
11      2048
12      4096
```

Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

## 10.8 Common Iteration Patterns

In Python there are two statements for iteration: the `for` statement and the `while` statement. The `for` statement is limited to those situations where we want to iterate for a fixed number of times or iterate over elements of a data structure of known size (like a string). The `while` statement is for every other situation. As a consequence, there are many `while` patterns. We have already seen some counting patterns. Only a few more are discussed here.

### 10.8.1 Basic While Pattern

The most basic (and most general) `while` pattern is as follows:

```
# initialise variables related to <condition>
while <condition>:
    # do some processing
    # potentially update variables related to <condition>
```

All `while` loops look like this. Most importantly, you must ensure that there is a chance the condition will change through each iteration, otherwise, there could be an infinite loop (the program keeps going until the heat death of the universe).

### 10.8.2 External Conditions

When retrieving data on demand (for example, from a user), it is often necessary to verify that the data is valid before continuing. The `while` loop is good for this case. The pattern looks like this:

```
keep_looping = True
while keep_looping:
    # read data from somewhere
    if <data is valid>:
        keep_looping = False
# do something with data
```

For example, to get a user to enter a mobile phone number and check that the number is valid, we might do the following:

```
number_not_valid = True
while number_not_valid:
    mobile_number = input('Enter your mobile phone number: ')
    if len(mobile_number)==10 and mobile_number.isdigit():
        number_not_valid = False
    else:
        print('There is an error in your number.')
print('Your mobile number is: ', mobile_number)
```

Or a simple guessing game function might look like this:

```
def guess_number(number):
    guess = int(input('Guess a number between 1 and 10: '))
    while guess != number:
        guess = int(input('Wrong. Guess again: '))
    print('Congratulations, you guessed right.')

guess_number(5)
```

If you also wanted to count the number of guesses needed, you could incorporate a counting pattern:

```
def guess_number(number):
    guess = int(input('Guess a number between 1 and 10: '))
    num_guesses = 1
    while guess != number:
        guess = int(input('Wrong. Guess again: '))
        num_guesses += 1
    print('Congratulations, you guessed right in ', num_guesses, 'guesses.')

guess_number(5)
```

### 10.8.3 Read one line at a time

Another file pattern involves reading one line at a time and doing something with each line as you go. This is especially useful if the file is very long as there is no need to read the whole file into memory, or you want to search for a specific thing in a file, and you don't want to read the whole file.

```
filename = 'words.txt'      # or whatever the filename is
wordfile = open(filename)   # first open the file
one_line = wordfile.readline() # reads a single line from the file
while one_line:
    # do something with oneline
    one_line = wordfile.readline()  # read the next line
wordfile.close()            # close the file
```

For example, to use readline to print out non-vowel characters:

```
filename = 'words.txt' # or whatever the filename is
wordfile = open(filename) # first open the file
one_line = wordfile.readline() # reads a single line from the file
while one_line:
    for ch in one_line: # this is an example of a nested loop
        if ch not in 'aeiouAEIOU':
            print(ch, end = "")
    one_line = wordfile.readline() # read the next line
wordfile.close() # close the file
```

### 10.8.4 Doctest escape character

In doctests, use \\n to specify the format for an expected \n being returned. The first backslash is an escape for the second, so stops the interpretation happening within the doctest. Notice that a returned string is formatted differently from a printed string in the doctest.

```
def ret_test(s):
    """
    >>> ret_test("dog")
    'dog\\ndog'
    """
    return s + '\n' + s

def prt_test(s):
    """
    >>> prt_test("cat")
    cat
    cat
    """
    print(s + "\n" + s)

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

## 10.9 Glossary

**multiple assignment:** Making more than one assignment to the same variable during the execution of a program.

**initialization (of a variable):** To initialize a variable is to give it an initial value, usually in the context of multiple assignment. Since in Python variables don't exist until they are assigned values, they are initialized when they are created. In other programming languages this is not the case, and variables can be created without being initialized, in which case they have either default or *garbage* values.

**increment** Both as a noun and as a verb, increment means to increase by 1.

**decrement** Decrease by 1.

**iteration:** Repeated execution of a set of programming statements.

**loop:** A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

**infinite loop:** A loop in which the terminating condition is never satisfied.

**trace:** To follow the flow of execution of a program by hand, recording the change of state of the variables and any output produced.

**counter** A variable used to count something, usually initialized to zero and incremented in the body of a loop.

**loop body:** The statements inside a loop.

**loop variable:** A variable used as part of the terminating condition of a loop.

## 10.10 Laboratory exercises

There are a lot of exercises for this lesson. If you feel like you've mastered the concepts, then there is no need to do all of them. However, they are all good practice for the mastery progressions and the practical test.

**When starting to write loops it can be easy to make a mistake and end up in an infinite loop. If you find yourself in this situation you can exit from it by pressing *Ctrl-C*.**

1. Write a function which will take a string parameter and return a string with all the vowels replaced by "_".

   Make a text file with a series of words on the same line e.g. Hubble bubble toil and trouble.

   In the main routine, write code to

   (a) open the file

   (b) read the contents to a string

   (c) split the string into a list of "words"

   (d) use a `for` loop to send each word to the function you wrote and display the string that is returned.

   Output:

```
H_bbl_
b_bbl_
t__l
_nd
tr__bl_
```

2. (a) Write a function which will take a string parameter and returns that string with a "$" sign before it and ",000" after it. Make a text file and populate it with a series of integers e.g. 35 15 353 1 30 54 44 501.

   In the main routine, write code to open the file, read the contents to a string, split the string into a list of "words", use a `for` loop to send each word to the function you wrote, and print the result.

   (b) Add K to some of the integers of the text file e.g. 35K 15 353K 1K 30 54K 44K 501. Adapt the function so that if a "K" is part of the number, ",000" is appended as before after removing the "K". If "K" is not part of the number, then simply return the number. For example, a file with the above contents would result in:

```
$35,000
$15
$353,000
$1,000
$30
$54,000
$44,000
$501
```

3. Recall `num_digits` from page 127. What will `num_digits(0)` return? Modify it to return `1` for this case. Modify `num_digits` so that it works correctly with any integer value. Type the following into a script.

```
def num_digits(n):
    """
    >>> num_digits(12345)
    5
    >>> num_digits(0)
    1
    >>> num_digits(-12345)
    5
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Add your function body to `num_digits` and confirm that it passes the doctests, by running it.

### Show your work to a demonstrator here to make sure you are on the right track.

4. Add the following to your script.

```
def num_even_digits(n):
    """
    >>> num_even_digits(123456)
    3
    >>> num_even_digits(2468)
    4
    >>> num_even_digits(1357)
    0
    >>> num_even_digits(2)
    1
    >>> num_even_digits(20)
    2
    """
```

Write a body for `num_even_digits` and run it to check that it works as expected.

5. Add the following to your program:

```
def print_digits(n):
    """
    >>> print_digits(13789)
    9 8 7 3 1
    >>> print_digits(39874613)
    3 1 6 4 7 8 9 3
    >>> print_digits(213141)
    1 4 1 3 1 2
    """
```

Write a body for `print_digits` so that it passes the given doctests. In this case you can use the `print` function rather than a return statement if you like. Remember giving the `print` function's `end` argument a value of `' '` or, if you prefer the digits unspaced, `''` will stop the cursor from moving to a new line.

6. Write a function which takes two arguments, a string and a target letter. Determine and return the encrypted word. The encrypted word is made up of the characters after each target letter, in order. Here are the doctests:

```
    """
    >>> decode("I want no more to endure the names I am called by many people", 'n')
    'today'
    >>> decode("Hungry hares in lairs","r")
    'yes'
    >>> decode("Turn around dear","r")
    'no'
    """
```

To pass the 3rd doctest, you will need to check that the search has not gone on past the end of the string.

7. Write a function which takes two arguments, a filename and an integer representing line-length. Print the text of the file out in lines of the length specified. In the output below, the spaces are specified using the '_' character so you can see them, but they should be actual spaces in the doctest in your code. Ignore end-of-line characters (\n). How will you deal with the last line? For a file called words.txt containing:

```
I have a bunch
of red roses
```

here are example doctests:

```
    """
    >>> line_length("words.txt",7)
    I_have_
    a_bunch
    of_red_
    roses
    """
```

8. Create a file called `id_file.txt` containing the following lines:

```
123,15,2017
4264,25,2003
2014,22,1998
4721,16,2017
```

Each line of a file contains 3 items, ID, age and year. Each item is separated by a comma. Read the file and display the data separated by space. Describe the data with a header row. The doctests are:

```
    """
    >>> id_format("id_file.txt")
    ID age year
    123 15 2017
    4264 25 2003
    2014 22 1998
    4721 16 2017
    """
```

9. Use the same file from the previous question. Read the file. Count how many lines match the target year (last item). The doctests are:

```
    """
    >>> id_format3("id_file.txt",2017)
    2
    """
```

10. Write a function which takes a filename as an argument. Print the first word of each line in the file. Doctest example using words.txt from question 7:

```
    """
    >>> print_first_word("words.txt")
    I of
    """
```

**Show your work to a demonstrator here to make sure you are on the right track.**

11. Write a function which takes a filename as an argument. For each line in the file, print the first word, and then print a "w" for any other word. Doctest example using words.txt from question 7:

```
    """
    >>> print_line3("words.txt")
    I w w w
    of w w
    """
```

12. Write a function which takes a filename and a character as arguments. Print any line which begins with the character. (Create your own file. Don't worry about a doctest.)

13. For this question, create a file called `fishy.txt` that contains the lines:

```
One Fish
Two Fish
Red Fish
Blue Fish
```

    (a) Print the file with the newline character removed from every odd-numbered line. For example:

```
>>> compact_a("fishy.txt")
One fishTwo fish
Red fishBlue fish
```

    (b) Create a function that removes the newline character from every odd-numbered line (you may already have one from the previous question). Return a string with all the output in one string. Here is a doctest:

```
    """
    >>> compact_b("fishy.txt")
    'One fishTwo fish\\nRed fishBlue fish\\n'
    """
```

    You can tell clearly from the doctests for `compact_b` that the cursor in the text file is expected to have gone down after the line containing 'Blue fish'.

14. Create a file called numbers.txt that has an integer on every line. Modify the last few lines of your script from Questions 3: comment out the doctest lines; and add lines which read data from a file:

```
if __name__ == "__main__":
    # import doctest
    # doctest.testmod(verbose=True)
    numfile = open("numbers.txt")
    line = numfile.readline()
    while line:
        print(line, end='')
        line = numfile.readline()
    numfile.close()
```

Unlike in the last chapter, when we read an entire file in at once using `read` or `readlines`, here we are reading one line at a time using the `readline` method. When we reach the end of the file then `while line:` will evaluate to `False` and our loop will finish. Also notice the use of the "`end`" argument to prevent an extra newline from being added after each line.

Make the changes above to your script and run it to confirm that it prints out the contents of the file that gets read. Note that the file (`numbers.txt`) should be in the same directory that your script file is saved in.

15. Add code to your script to call `num_digits` on each line of `'numbers.txt'` so that instead of printing each line of the file it prints out the number of digits found on each line of the file. *Hint:* Remember what type of data `readline` returns and what type of data `num_digits` requires.

16. Write and doctest a function `add_all` which takes 2 integer parameters and returns the sum of all the integers between (and including) the first and second integer e.g. `add_all(4,6)` would return 15 (the sum 4 + 5 + 6 ). Think about what doctests might be useful.

17. *Extension Exercise:* Open a new script file and add the following:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

Write a function, `is_prime`, which takes a single integer argument and returns `True` when the argument is a **prime number** and `False` otherwise. Add doctests to your function as you develop it.

18. *Extension Exercise:* Write a function that prints all the prime numbers less than a given number:

```
def prime_numbers(n):
    """
    >>> prime_numbers(10)
    2 3 5 7
    >>> prime_numbers(20)
    2 3 5 7 11 13 17 19
    """
```

## 10.11 Mastery Level 6 - Practice Questions

Mastery Level 6 tests your understanding of material covered in lesson 10, specifically simple while loops.

There are 2 questions in the test.

You will need to download support files - these are available on Blackboard. Don't alter the files - they are needed for the doctests to work.

You should check your solutions with a demonstrator.

**While loops to generate sequences or get valid input from user**

1. Write a function that takes an int. The function should ask the user to enter a word longer than the int and continue asking until the user successfully does so. It should then return the word.

   Remember to have all the required files in the same directory as the python file.

   The function should offer sensible feedback to the user and be able to produce the behaviour shown below:

```
>>> word = get_word_longer_than(12)
Please enter a word longer than 12: this is not a word
Try again
Please enter a word longer than 12: 1234567891234567
Try again
Please enter a word longer than 12: supralapsarianism
>>> word
'supralapsarianism'
>>>
```

2. Write a function that asks the user for their preferred unit of mass until a valid one is entered and then returns it. Acceptable values are: 'kg', 'lb', 'g', 'oz'.

   The function should use a while loop.

   The function should offer sensible feedback to the user and be able to produce the behaviour shown below:

```
>>> unit_type = get_mass_unit()
Please enter your preferred mass measurement (kg, lb, g or oz): mg
Sorry, mg is not a valid choice
Please enter your preferred mass measurement (kg, lb, g or oz): tonne
Sorry, tonne is not a valid choice
Please enter your preferred mass measurement (kg, lb, g or oz): oz
>>> unit_type
'oz'
>>>
```

3. Write a function that takes 3 int arguments x, y and limit. It should return a string containing all the positive numbers that are multiples of x but not multiples of y, up to and including limit. The numbers should each be followed by a space.

   The function should use a while loop.

   The function should pass the doctests shown below:

```
def print_multiples(x, y, limit):
    '''
    >>> print_multiples(3, 4, 30)
    '3 6 9 15 18 21 27 30 '
    >>> print_multiples(5, 7, 90)
    '5 10 15 20 25 30 40 45 50 55 60 65 75 80 85 90 '
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

4. Write a function that takes an int argument, x, and returns a string containing the numbers from the Fibonacci series (starting from 1) that are less than x. Each number should be followed by a space.

   Remember: the first two numbers in the Fibonacci sequence are (in our case) 1 and 1, and each subsequent number is the sum of the previous two.

   1 1 2 3 5 8 13 . . .
   The function should use a while loop.

   The function should pass the doctests shown below:

```
def fibonacci(x):
    '''
    >>> fibonacci(10)
    '1 1 2 3 5 8 '
    >>> fibonacci(300)
    '1 1 2 3 5 8 13 21 34 55 89 144 233 '
    >>> fibonacci(377)
    '1 1 2 3 5 8 13 21 34 55 89 144 233 '
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

**While loops to process files**

1. Write a function that takes 2 arguments - a file name and an int, x. The function should, using a while loop, return a string containing the first x vowels found in the file. You may assume the file is not overly large. If there are fewer than x vowels the function should simply return a string containing all the vowels found.

   Remember to have all the required files in the same directory as the python file.

   Your function should pass the doctests below:

```
def first_x_vowels(file_name, x):
    """
    >>> first_x_vowels('test.txt', 10)
    'aeiou'
    >>> first_x_vowels('helen_clark.txt', 50)
    'eeEiaeaOIoeuaiaeeaaoiiiaoeeaeieiieoeeaaooaaeAiiaoo'
    >>> first_x_vowels('peter_fraser.txt', 15)
    'eeaeAuueeeaaeea'
    """

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

2. Write a function that takes a string argument representing a filename and an int argument, cut_off.

   The function should then open the file and, using a while loop, return a string containing the index numbers of words that have a length greater than the cut_off number. Each index number should be followed by a space.

   Remember: the first word is at index 0.

   Remember to have all the required files in the same directory as the python file.

   Note: You do not have to strip punctuation. Commas and full stops etc are just treated as a part of the word.

   Your function should pass the doctests below:

```
def positions_of_words_longer_than(filename, cut_off):
    '''
    >>> positions_of_words_longer_than('richard_seddon.txt', 12)
    '46 197 565 692 825 840 '
    >>> positions_of_words_longer_than('helen_clark.txt', 12)
    '32 47 155 235 315 412 419 424 524 659 791 796 819 837 884 885 974 1137 '
    >>> positions_of_words_longer_than('robert_muldoon.txt', 12)
    '85 175 190 212 244 284 520 716 858 909 910 989 1003 '
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

3. Write a function to open a file containing digits. Your function should return the highest product of any 3 consecutive numbers in the file. If there are fewer than three numbers in the file the function should return zero. You may assume that the file contains only digits.

   Remember to have all the required files in the same directory as the python file.

   Your function should pass the following doctests:

```
def highest_product(file_name):
    '''
    >>> highest_product('test.txt')
    120
    >>> highest_product('digits_4.txt')
    648
    >>> highest_product('digits_3.txt')
    729
    >>> highest_product('test_2.txt')
    0
    '''

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# A note on the rest of the content and mastery levels

The content and mastery levels up to now (Mastery Level 6) can be considered foundational (input, output, math, conditionals, iteration). These concepts and the application of them are the minimum required to have any understanding of programming. Everything else is basically built on top.

Nevertheless, there are other topics that could also be considered foundational and it is very hard to do programming without them. The most foundational of these is an understanding of data structures, and the most fundamental data structure in Python is the list. All remaining mastery levels assume some basic understanding of lists. It is worthwhile (even necessary) to have a decent understanding of the content of Chapters 12 and 13 for the other mastery levels.

# Lesson 11

# Iteration: part 2

## 11.1 Two-dimensional tables

In this lesson we look at nested loops. An illustrative example of nested loops is printing out two-dimensional tables. A two-dimensional table is a table where you read the value at the intersection of a row and a column. A multiplication table is a good example. The code below prints a multiplication table that goes up to $3 \times 3$:

```
def multiplication_table(limit):
    row = 1
    while row <= limit:
        column = 1
        while column <= limit:
            print (row * column, end='\t')
            column += 1
        print()
        row += 1

multiplication_table(3)
```

Nested loops can certainly be quite challenging to read (and write) but see if you can follow what is happening. Notice that we have placed a while loop inside another. The outer loop counts `row` from 1 through to limit (in this case 3). Each time we enter the outer loop, the inner loop counts `column` from 1 through to limit (in this case 3).

If we trace the program we can understand what is happening. Each row of the following table represents a change in the output. Values of the variables row and column are also shown:

| row | column | total output |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 1 2 |
| 1 | 3 | 1 2 3 |
| 2 | 1 | 1 2 3<br>2 |
| 2 | 2 | 1 2 3<br>2 4 |
| 2 | 3 | 1 2 3<br>2 4 6 |
| 3 | 1 | 1 2 3<br>2 4 6<br>3 |
| 3 | 2 | 1 2 3<br>2 4 6<br>3 6 |
| 3 | 3 | 1 2 3<br>2 4 6<br>3 6 9 |

That was a lot of code to produce in one go. Let's start over with a much more incremental and sensible approach. Let's say you want to print a multiplication table for the values from 1 to 6. A good way to start is to write a loop that prints the multiples of 2, all on one line:

```
i = 1
while i <= 6:
    print(2*i, end='   ')
    i += 1
print()
```

The first line initializes a variable named `i`, which acts as a counter or **loop variable** . As the loop executes, the value of `i` increases from 1 to 6. When `i` is 7, the loop terminates. Each time through the loop, it displays the value of `2*i`. Again, the `end` argument replaces the newline, in this case with three spaces. After the loop completes, the second print call starts a new line. The output of the program is:

```
2   4   6   8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize** .

## 11.2 Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a function, allowing you to take advantage of all the things functions are good for. You have already seen two examples of encapsulation: `print_parity` in Lecture 5; and `is_divisible` in Lecture 6. Generalization means taking something specific, such as printing the multiples of 2, and making it more general, such as printing the multiples of any integer. This function encapsulates the previous loop and generalizes it to print multiples of `n` separated by a tab:

```
def print_multiples(n):
    i = 1
    while i <= 6:
        print (n*i, end='\t')
        i += 1
    print()
```

To encapsulate, all we had to do was add the first line, which declares the name of the function and the parameter list. To generalize, all we had to do was replace the value 2 with the parameter `n`. If we call this function with the argument `2`, we get the same output as before. With the argument `3`, the output is:

```
3       6       9       12      15      18
```

With the argument `4`, the output is:

```
4       8       12      16      20      24
```

By now you can probably guess how to print a multiplication table—by calling `print_multiples` repeatedly with different arguments. In fact, we can use another loop:

```
i = 1
while i <= 6:
    print_multiples(i)
    i += 1
```

Notice how similar this loop is to the one inside `print_multiples`. All we did was replace the print statement with a function call.

The output of this program is a multiplication table:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

This is an example of a nested loop where one loop is *nested* inside another. In this case, the inner loop is defined inside a function, and that is often useful for understanding what is going on, but is not necessary. Nested loops are discussed more in Section 11.9.1.

## 11.3 More encapsulation

To demonstrate encapsulation again, let's take the code from the last section and wrap it up in a function:

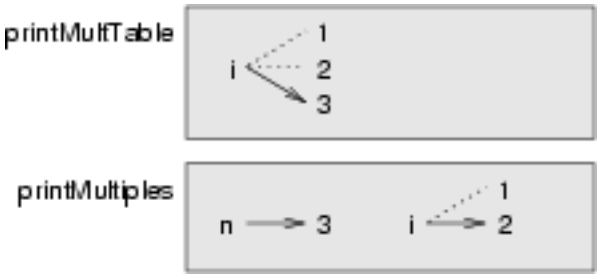
```
def print_mult_table():
    i = 1
    while i <= 6:
        print_multiples(i)
        i += 1
```

This process is a common **development plan**. We develop code by writing lines of code outside any function, or typing them in to the interpreter. When we get the code working, we extract it and wrap it up in a function. This development plan is particularly useful if you don't know how to divide the program into functions when you start writing. This approach lets you design as you go along.

## 11.4 Local variables

You might be wondering how we can use the same variable, `i`, in both `print_multiples` and `print_mult_table`. Doesn't it cause problems when one of the functions changes the value of the variable? The answer is no, because the `i` in `print_multiples` and the `i` in `print_mult_table` are *not* the same variable. Variables created inside a function definition are **local**; you can't access a local variable from outside its home function. That means you are free to have multiple variables with the same name as long as they are not in the same function.

The stack diagram for this program shows that the two variables named `i` are not the same variable. They can refer to different values, and changing one does not affect the other.




The value of `i` in `print_mult_table` goes from 1 to 6. In the diagram it happens to be 3. The next time through the loop it will be 4. Each time through the loop, `print_mult_table` calls `print_multiples` with the current value of `i` as an argument. That value gets assigned to the parameter `n`. Inside `print_multiples`, the value of `i` goes from 1 to 6. In the diagram, it happens to be 2. Changing this variable has no effect on the value of `i` in `print_mult_table`.

It is common and perfectly legal to have different local variables with the same name. In particular, names like `i` and `j` are used frequently as loop variables. If you avoid using them in one function just because you used them somewhere else, you will probably make the program harder to read.

## 11.5 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `print_mult_table`:

```
def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i)
        i += 1
```

We replaced the value 6 with the parameter high. If we call print_mult_table with the argument 7, it displays:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

This is fine, except that we probably want the table to be square - with the same number of rows and columns. To do that, we add another parameter to print_multiples to specify how many columns the table should have. Just to be annoying, we call this parameter high, demonstrating that different functions can have parameters with the same name (just like local variables). Here's the whole program:

```
def print_multiples(n, high):
    i = 1
    while i <= high:
        print(n*i, end = '\t')
        i += 1
    print()


def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i, high)
        i += 1
```

Notice that when we added a new parameter, we had to change the first line of the function (the function heading), and we also had to change the place where the function is called in print_mult_table. As expected, this program generates a square seven-by-seven table:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

When you generalize a function appropriately, you often get a program with capabilities you didn't plan. For example, you might notice that, because $ab = ba$, all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of print_mult_table. Change

```
print_multiples(i, high)
```

to

```
print_multiples(i, i)
```

and you get

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

## 11.6 Functions

A few times now, we have mentioned all the things functions are good for. By now, you might be wondering what exactly those things are. Here are some of them:

1. Giving a name to a sequence of statements makes your program easier to read and debug.

2. Dividing a long program into functions allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.

3. Facilitating the use of iteration.

4. Producing reusable code; as well defined functions are often useful in many different situations.

## 11.7 Newton's method

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it. For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of $n$. If you start with almost any approximation, you can compute a better approximation with the following formula[1]:

```
better = (approx + n/approx)/2
```

By repeatedly applying this formula until the better approximation is equal to the previous one, we can write a function for computing the square root:

```
def sqrt_newton(n):
    eps = 1e-8
    approx = n/2.0
    better = (approx + n/approx)/2.0
    while abs(better-approx) > eps:
        approx = better
        better = (approx + n/approx)/2.0
    return approx
```

Try calling this function with 25 as an argument to confirm that it returns 5.0.

[1] If $s$ is the square root of $n$, then $s$ and $n/s$ are the same, so even for a number $x$ which isn't the square root, the average of $x$ and $n/x$ might well be a better approximation.

## 11.8 Algorithms

Newton's method is an example of an **algorithm** : it is a mechanical process for solving a category of problems (in this case, computing square roots). It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic. But if you were lazy, you probably cheated by learning a few tricks. For example, to find the product of $n$ and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm! Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

Executing algorithms does not require any great intelligence but does require great attention to detail, and that's why computers are so good at it. On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming. Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Unfortunately, there is no algorithm for designing algorithms! But there are methods that can make it easier. Test-driven development is one of those. If you're not sure how to solve a particular problem, try going back to look at Section 7.4 to reboot your thinking.

## 11.9 Common Iteration Patterns

### 11.9.1 Nested Loops

Nested loops can be tricky, but the common pattern is that we need to enumerate a bunch of things, and for each of those things, we need to enumerate a bunch of other things. For example, let's say your grandparents want to write down all of their grandchildren. They might first list each of their children, and then for each child, list each of their child's children. The general pattern using a while loop looks something like this:

```
# initialise outer loop condition
while <outer condition>:
    # initialise inner loop condition
    while <inner condition>:
        # do some processing
        # update variables related to <inner condition>
    # update variables related to <outer condition>
```

Sometimes it easier to understand if the inner while loop is put into a function as we did above.

For example, if we want to find all factors of all the numbers from 1 to 100. First we decide how to find all the factors of one number (this is the inner loop). The easiest way is to check all the numbers up to the number of interest. This whole thing needs to be repeated for all the numbers up to 100 (the outer loop). We end up with:

```
number = 1
while number <= 100: # check all numbers from 1 to 100
    possible_factor = 1
    print(number, end = ": ")
    while possible_factor<number:  # check all numbers up to number
        if number % possible_factor == 0:  # we've found a factor
            print(possible_factor, end=' ')
        possible_factor += 1       # update the condition variable
    print()
    number += 1
```

Can you generalise this code to find only prime factors?

### 11.9.2 Counting

We've seen some counting patterns before and these keep coming up over and over again. The general pattern for while loops is:

```
count = 0
while <condition>:
    count = count + 1
    # update variables associated with condition
```

For example, suppose you wanted a function to compute the number of years it would take to reach a certain target if you invested some base amount of money into an account that compounded interest each year. That function might look like:

```
def how_many_years(principal, interest_rate, target):
    balance = principal # initial balance of account
    years = 0 # start counting
    while balance < target:  # keep going until target met
        balance = balance + interest_rate*balance # add the interest back in
        years += 1
    return years


print(how_many_years(100, 0.05, 1000))
print(how_many_years(100, 0.05, 10000))
print(how_many_years(100, 0.05, 100000))
print(how_many_years(100, 0.05, 1000000))
```

Try it out to see the power of compound interest.

### 11.9.3 Convergence

Newton's sqrt algorithm is a good example of a convergence pattern. Convergence patterns occur very frequently in numerical algorithms. The basic pattern is:

```
eps = # some acceptable margin
estimate = # estimate answer
old_estimate = # something different to estimate
while abs(estimate-old_estimate)>eps:  # keep going until there is no change
    old_estimate = estimate # keep the last estimate
    estimate = # update estimate
```

The trick is finding an update rule that brings you closer to the actual solution.

## 11.10 Glossary

**tab:** A special character that causes the cursor to move to the next tab stop on the current line.

**newline:** A special character that causes the cursor to move to the beginning of the next line.

**cursor:** An invisible marker that keeps track of where the next character will be printed.

**escape sequence:** An escape character, \, followed by one or more printable characters used to designate a non-printable character.

**encapsulate:** To divide a large complex program into components (like functions) and isolate the components from each other (by using local variables, for example).

**generalize:** To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

**development plan:** A process for developing a program. In this chapter, we demonstrated a style of development based on developing code to do simple, specific things and then encapsulating and generalizing.

**algorithm:** A step-by-step process for solving a category of problems.

## 11.11 Laboratory Exercises

1. Write a program that prompts the user for some target string (e.g. "daffodils"), and a filename (e.g. "mary.txt"). Open the file and read the contents using `readlines` (make sure the file exists - either use an existing file or create a new one using a text editor). Print out the lines that match the target string and count how many lines match.

   Modularise your code, so that the search string is an input to a function.

2. Make a text file containing integer values. Create a program that: reads the contents of the file using the `read` method; creates a list of all the numbers using the `split` method; uses a loop to compute the sum of all the integers in the file.

### Show your work to a demonstrator here to make sure you are on the right track.

3. (a) Create a function, called `print_powers`, that takes two parameters, $n$ and `high`, and prints the values of $n^1, n^2, \cdots, n^{\text{high}}$ on a single line.

   (b) Create another function, called `print_powers_table`, that takes a single parameter, `high`, and prints on the first line the result of $1^1, 1^2, \cdots, 1^{\text{high}}$, then on the second line the result of $2^1, 2^2, \cdots, 2^{\text{high}}$, and so on until the last line is printed as $\text{high}^1, \text{high}^2, \cdots, \text{high}^{\text{high}}$.

4. (a) Write a function, `is_prime`, which takes a single integer argument, $n$, and returns `True` when the argument is a **prime number** and `False` otherwise. Add doctests to your function as you develop it. Use doctests to test your function. Hint: A simple method is to loop through all the numbers from 2 to $n/2$ and check if any of the numbers divide $n$ evenly (i.e. no remainder).

   (b) Write a function that finds all the prime numbers less than a given number. Use doctests to test your function.

5. *Extension Exercise:* Write a spell-checking program that reads in a text file, scans through every word in the file and checks that it is spelled correctly. To do this you will probably also want to read in the system dictionary: `/usr/share/dict/words`.

### Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 12

# Lists part 1

A **list** is an ordered set of values, where each value is identified by an index. The values that make up a list are called its **elements** . Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings—and other things that behave like ordered sets—are called **sequences** .

## 12.1 List values

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (`[` and `]`):

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list:

```
["hello", 2.0, 5, [10, 20]]
```

A list within another list is said to be **nested** . Finally, there is a special list that contains no elements. It is called the empty list, and is denoted `[]`. Like numeric 0 values and the empty string, the empty list is false in a boolean expression:

```
>>> if []:
...     print("This is true.")
... else:
...     print("This is false.")
...
This is false.
>>>
```

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as parameters to functions. We can:

```
>>> vocabulary = ["ameliorate", "castigate", "defenestrate"]
>>> numbers = [17, 5]
>>> empty = []
>>> print(vocabulary, numbers, empty)
['ameliorate', 'castigate', 'defenestrate'] [17, 5] []
```

## 12.2 Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string—the bracket operator (`[]` – not to be confused with an empty list). The expression inside the brackets specifies the index. Remember that the indices start at `0`:

```
>>> print(numbers[0])
17
```

Any integer expression can be used as an index:

```
>>> numbers[9-8]
5
>>> numbers[1.0]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

If you try to read or write an element that does not exist, you get a runtime error:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
IndexError: list index out of range
```

If an index has a negative value, it counts backward from the end of the list:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
IndexError: list index out of range
```

`numbers[-1]` is the last element of the list, `numbers[-2]` is the second to last, and `numbers[-3]` doesn't exist. It is common to use a loop variable as a list index.

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
   print(horsemen[i])
   i += 1
```

This while loop counts from 0 to 4. When the loop variable `i` is 4, the condition fails and the loop terminates. The body of the loop is only executed when `i` is 0, 1, 2, and 3. Each time through the loop, the variable `i` is used as an index into the list, printing the $i^{th}$ element. This pattern of computation is called a **list traversal**.

## 12.3 List length

The function `len` returns the length of a list, which is equal to the number of its elements. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
num = len(horsemen)
while i < num:
   print(horsemen[i])
   i += 1
```

The last time the body of the loop is executed, `i` is `len(horsemen) - 1`, which is the index of the last element. When `i` is equal to `len(horsemen)`, the condition fails and the body is not executed, which is a good thing, because `len(horsemen)` is not a legal index.

Although a list can contain another list, the nested list still counts as a single element. The length of this list is 4:

```
["spam!", 1, ["Brie", "Roquefort", "Pol le Veq"], [1, 2, 3]]
```

## 12.4 The range type

Lists that contain consecutive integers are common, so Python provides the **range** type to represent immutable sequences of numbers. The **range** type generates the numbers as needed which is more memory efficient. A range can be converted into a list so we can better see the sequence produced:

```
>>> range(2, 9)
range(2, 9)
>>> list(range(2, 9))
[2, 3, 4, 5, 6, 7, 8]
```

The range takes two arguments and creates a sequence that contains all the integers from the first to the second, including the first but *not the second*.

There are two other forms of range. With a single argument, it creates a range that starts at 0:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If there is a third argument, it specifies the space between successive values, which is called the **step size**. This example counts from 1 to 10 by steps of 2:

```
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
```

If the step size is negative, then start must be greater than stop:

```
>>> list(range(20, 4, -5))
[20, 15, 10, 5]
```

or the result will be an empty list:

```
>>> list(range(10, 20, -5))
[]
```

## 12.5 Lists and for loops

The **`for`** loop also works with lists and ranges. The generalized syntax of a `for` loop is:

```
for VARIABLE in LIST: # or for VARIABLE in RANGE:
   BODY
```

This statement is equivalent to:

```
i = 0
while i < len(LIST):
   VARIABLE = LIST[i]
   BODY
   i += 1
```

The `for` loop is more concise because we can eliminate the loop variable, `i`. Here is the `horsemen while` loop written with a `for` loop:

```
for horseman in horsemen:
   print(horseman)
```

It almost reads like English: For (every) `horseman` in (the list of) `horsemen`, print (the name of the) `horseman`.

Any list expression or range can be used in a for loop:

```
for number in range(20):
   if number % 3 == 0:
       print (number)

for fruit in ["banana", "apple", "quince"]:
   print("I like to eat " + fruit + "s!")
```

The first example prints all the multiples of 3 between 0 and 19. The second example expresses enthusiasm for various fruits.

Since lists are mutable, it is often desirable to traverse a list, modifying each of its elements. The following squares all the numbers from 1 to 5:

```
numbers = [1, 2, 3, 4, 5]

for index in range(len(numbers)):
   numbers[index] = numbers[index]**2
```

Take a moment to think about `range(len(numbers))` until you understand how it works. We are interested here in both the *value* and its *index* within the list, so that we can assign a new value to it.

This pattern is common enough that Python provides a nicer way to implement it:

```
numbers = [1, 2, 3, 4, 5]

for index, value in enumerate(numbers):
   numbers[index] = value**2
```

`enumerate` generates both the `index` and the `value` associated with it during the list traversal. Try this next example to see more clearly how enumerate works:

```
>>> for index, value in enumerate(["banana", "apple", "pear", "quince"]):
...     print(index, value)
...
0 banana
1 apple
2 pear
3 quince
>>>
```

## 12.6 List membership

`in` is a boolean operator that tests membership in a sequence. We used it previously with strings, but it also works with lists and other sequences:

```
>>> horsemen = ["war", "famine", "pestilence", "death"]
>>> "pestilence" in horsemen
True
>>> "debauchery" in horsemen
False
```

Since `pestilence` is a member of the `horsemen` list, the `in` operator returns `True`. Since `debauchery`

is not in the list, `in` returns `False`. We can use `not` combined with `in` to test whether an element is not a member of a list:

```
>>> "debauchery" not in horsemen
True
```

## 12.7 List operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

Similarly, the $\star$ operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats `[0]` four times. The second example repeats the list `[1, 2, 3]` three times.

Note: ranges cannot be concatenated or multiplied but you can, of course convert them to lists and proceed as above:

```
>>> list(range(0, 10, 2)) + list(range(11, 20, 2))
[0, 2, 4, 6, 8, 11, 13, 15, 17, 19]
>>> list(range(3)) * 3
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

## 12.8 List slices

The slice operations we saw with strings also work on lists:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

## 12.9 Lists are mutable

Unlike strings, lists are **mutable** , which means we can change their elements. Using the bracket operator on the left side of an assignment, we can update one of the elements:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print(fruit)
['pear', 'apple', 'orange']
```

The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of `fruit` has been changed from `"banana"` to `"pear"`, and the last from `"quince"` to `"orange"`. An assignment to an element of a list is called **item assignment**.

Item assignment does not work for strings:

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

but it does for lists:

```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

With the slice operator we can update several elements at once:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = ["x", "y"]
>>> print(a_list)
['a', 'x', 'y', 'd', 'e', 'f']
```

We can also remove elements from a list by assigning the empty list to them:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> a_list[1:3] = []
>>> print(a_list)
['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location:

```
>>> a_list = ["a", "d", "f"]
>>> a_list[1:1] = ["b", "c"]
>>> print(a_list)
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ["e"]
>>> print(a_list)
['a', 'b', 'c', 'd', 'e', 'f']
```

## 12.10 List deletion

Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable. `del` removes an element from a list:

```
>>> a = ["one", "two", "three"]
>>> del a[1]
>>> a
['one', 'three']
```

As you might expect, `del` handles negative indices and causes a runtime error if the index is out of range.

You can use a slice as an index for del:

```
>>> a_list = ["a", "b", "c", "d", "e", "f"]
>>> del a_list[1:5]
>>> print(a_list)
['a', 'f']
```

As usual, slices select all the elements up to, but not including, the second index.

## 12.11 Glossary

**list:** A named collection of objects, where each object is identified by an index.

**range:** An immutable sequence of numbers often used in for loops

**index:** An integer variable or value that indicates an element of a list.

**element:** One of the values in a list (or other sequence). The bracket operator selects elements of a list.

**sequence:** Any of the data types that consist of an ordered set of elements, with each element identified by an index.

**nested list:** A list that is an element of another list.

**step size:** The interval between successive elements of a linear sequence. The third (and optional argument) when creating a range is called the step size. If not specified, it defaults to 1.

**list traversal:** The sequential accessing of each element in a list.

**mutable type:** A data type in which the elements can be modified. All mutable types are compound types. Lists are mutable data types; strings are not.

**object:** A thing to which a variable can refer.

## 12.12 Laboratory exercises

1. (a) Write a function that takes a list as an argument and returns the 5th element if it exists and "No such element" if it doesn't.

   (b) Write a function that takes a list as an argument and returns the last two elements of the list.

   (c) Write a function that takes a list as an argument and returns every second element of the list. Note: the returned value should also be a list.

2. Write a loop that traverses:

```
["spam!", [1], ["Brie", "Roquefort", "Pol le Veq"], [1, 2, 3]]
```

   and prints the length of each element.

3. What is the Python interpreter's response to the following?

```
>>> list(range(10, 0, -2))
```

   The three arguments when creating a `range` are `start`, `stop`, and `step`, respectively. In this example, `start` is greater than `stop`. What happens if `start < stop` and `step < 0`? Write a rule for the relationships among `start`, `stop`, and `step`.

### Show your work to a demonstrator here to make sure you are on the right track.

4. Write a function `add_lists(a, b)` that takes two lists of numbers of the same length, and returns a new list containing the sums of the corresponding elements of each. The original lists should remain unchanged.

```
def add_lists(a, b):
    """
    >>> add_lists([1, 1], [1, 1])
    [2, 2]
    >>> add_lists([1, 2], [1, 4])
    [2, 6]
    >>> add_lists([1, 2, 1], [1, 4, 3])
    [2, 6, 4]
    >>> list1 = [1, 2, 1]
    >>> list2 = [1, 4, 3]
    >>> sum = add_lists(list1, list2)
    >>> list1 == [1, 2, 1]
    True
    >>> list2 == [1, 4, 3]
    True
    """
```

   `add_lists` should pass the doctests above.

5. Write a function `mult_lists(a, b)` that takes two lists of numbers of the same length, and returns the sum of the products of the corresponding elements of each.

```
def mult_lists(a, b):
    """
    >>> mult_lists([1, 1], [1, 1])
    2
    >>> mult_lists([1, 2], [1, 4])
    9
    >>> mult_lists([1, 2, 1], [1, 4, 3])
    12
    """
```

   Verify that `mult_lists` passes the doctests above.

6. *Extension Exercise:* Write a function called `flatten_list` that takes as input a list which may be nested, and returns a non-nested list with all the elements of the input list.

```
def flatten_list(alist):
    '''
       >>> flatten_list([1,2,3])
       [1, 2, 3]
       >>> flatten_list([1, [2,3], [4, 5], 6])
       [1, 2, 3, 4, 5, 6]
    '''
```

   Is your solution general enough to allow any level of nesting? If not, can you make it so? Hint: you can use the `type` function to check if some variable is of a particular type. E.g. `if type(el) is list:`.

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 13

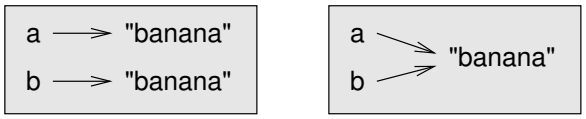
# Lists part 2

## 13.1 Objects and values

If we execute these assignment statements,

```
a = "banana"
b = "banana"
```

we know that `a` and `b` will refer to a string with the letters ``banana''. But we can't tell whether they point to the *same* string. There are two possible states:




In one case, `a` and `b` refer to two different things that have the same value. In the second case, they refer to the same thing. These things have names—they are called **objects**. An object is something a variable can refer to. Every object has a unique **identifier**, which we can obtain with the id function. By printing the identifier of `a` and `b`, we can tell whether they refer to the same object.

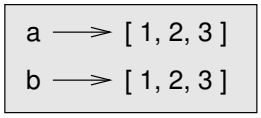
```
>>> id(a)
135044008
>>> id(b)
135044008
```

In fact, we get the same identifier twice, which means that Python only created one string, and both `a` and `b` refer to it. If you try this, your actual `id` value will probably be different. Python does this because strings are immutable and it is more space efficient to maintain only one copy of a string in memory.

Interestingly, lists behave differently. When we create two lists, we get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

So the state diagram looks like this:



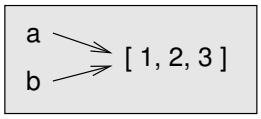

`a` and `b` have the same value but do not refer to the same object.

## 13.2 Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a) == id(b)
True
```

In this case, the state diagram looks like this:




Because the same list has two different names, `a` and `b`, we say that it is **aliased**. Changes made with one alias affect the other:

```
>>> b[0] = 5
>>> print(a)
[5, 2, 3]
```

Although this behaviour can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings when it sees an opportunity to economize.

## 13.3 Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy. The easiest way to clone a list is to use the slice operator:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print(b)
[1, 2, 3]
```

Taking any slice of `a` creates a new list. In this case the slice happens to consist of the whole list. Now we are free to make changes to `b` without worrying about `a`:

```
>>> b[0] = 5
>>> print(a)
[1, 2, 3]
```

## 13.4 List parameters

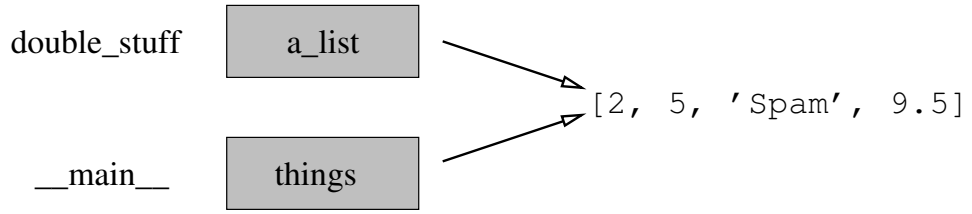
Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the parameter change the argument as well. For example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def double_stuff(a_list):
    for index, value in enumerate(a_list):
        a_list[index] = 2 * value
```

If we put `double_stuff` in a file named `double_stuff.py`, we can test it out like this:

```
>>> from double_stuff import double_stuff
>>> things = [2, 5, "Spam", 9.5]
>>> double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

The parameter `a_list` and the variable `things` are aliases for the same object. The state diagram looks like this:




Since the list object is shared by two frames, we drew it between them. If a function modifies a list parameter, the caller sees the change.

## 13.5 Pure functions and modifiers

Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**. A **pure function** does not produce side effects. It communicates with the calling program only through parameters, which it does not modify, and a return value. Here is `double_stuff` written as a pure function:

```
def double_stuff(a_list):
    new_list = []
    for value in a_list:
        new_list += [2 * value]
    return new_list
```

This version of `double_stuff` does not change its arguments:

```
>>> from double_stuff import double_stuff
>>> things = [2, 5, "Spam", 9.5]
>>> double_stuff(things)
[4, 10, 'SpamSpam', 19.0]
>>> things
[2, 5, "Spam", 9.5]
>>>
```

To use the pure function version of `double_stuff` to modify `things`, you would assign the return value back to `things`:

```
>>> things = double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

## 13.6 Which is better?

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, modifiers are convenient at times, and in some cases, functional programs are less efficient. In general, we recommend that you write pure functions whenever it is reasonable to do so and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

## 13.7 Nested lists

A nested list is a list that appears as an element in another list. In this list, the element with index 3 is a nested list:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

If we print `nested[3]`, we get `[10, 20]`. To extract an element from the nested list, we can proceed in two steps:

```
>>> elem = nested[3]
>>> elem[0]
10
```

Or we can combine them:

```
>>> nested[3][1]
20
```

Bracket operators evaluate from left to right, so this expression gets the third element of `nested` and extracts the first element from it.

## 13.8 Matrices

Nested lists are often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`matrix` is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

```
>>> matrix[1]
[4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
>>> matrix[1][1]
5
```

The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows.

## 13.9 Strings and lists

Python has a command called `list` that takes a sequence type as an argument and creates a list out of its elements.

```
>>> list("Crunchy Frog")
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

There is also a `str` command that takes any Python value as an argument and returns a string representation of it.

```
>>> str(5)
'5'
>>> str(None)
'None'
>>> str(list("nope"))
"['n', 'o', 'p', 'e']"
```

As we can see from the last example, `str` can't be used to join a list of characters together. To do this we could use the string method `join`:

```
>>> char_list = list("Frog")
>>> char_list
['F', 'r', 'o', 'g']
>>> "".join(char_list)
'Frog'
```

To go in the opposite direction, the `split` method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary:

```
>>> song = "The rain in Spain..."
>>> song.split()
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter (which doesn't appear in the result):

```
>>> song.split("ai")
['The r', 'n in Sp', 'n...']
```

## 13.10 Glossary

**modifier:** A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

**side effect:** A change in the state of a program made by calling a function that is not a result of reading the return value from the function. Side effects can only be produced by modifiers.

**pure function:** A function which has no side effects. Pure functions only make changes to the calling program through their return values.

**delimiter:** A character or string used to indicate where a string should be split.

**aliases:** Multiple variables that contain references to the same object.

**clone:** To create a new object that has the same value as an existing object. Copying a reference to an object creates an alias but doesn't clone the object.

## 13.11 Laboratory exercises

1. Consider the following code:

```
a = [1, 2, 3]
b = a[:]
b[0] = 5
```

Draw a state diagram for `a` and `b` before and after the third line is executed.

2.

```
song = "The rain in Spain..."
```

Describe the relationship between `" ".join(song.split())` and `song`. Are they the same for all strings? When would they be different?

3. What will be the output of the following program?

```
this = ["I", "am", "not", "a", "crook"]
that = ["I", "am", "not", "a", "crook"]
print("Test 1: {}".format(id(this) == id(that)))
that = this
print("Test 2: {}".format(id(this) == id(that)))
```

Provide a *detailed* explanation of the results.

4. Exercise 4 in Section 12.12 asked you to write a function to add two lists without modifying the original lists. Suppose you had written the following function (with a little bit of test code):

```
def add_lists(a, b):
    for i in range(len(a)):
        a[i] += b[i]
    return a

list1 = [1,2,3]
list2 = [3,4,6]
list3 = add_lists(list1, list2)
print(list1)
print(list2)
print(list3)
```

What is the output of the print statements and why? This is an example of a side-effect. How can we write our code to avoid side-effects?

**Show your work to a demonstrator here to make sure you are on the right track.**

5. A list contains pairs of items. The first of every pair is a string representing a surname / family name. The second item is a list containing one or more first names. Write a function which takes a list of this description as a parameter, and **prints out** all the full names. Here are some doctests:

```
def print_names(name_list):
    """
    >>> print_names(["Brown",["Bob","Betty"],"Green",["Geoff","Gemma","Gail"]])
    Bob Brown
    Betty Brown
    Geoff Green
    Gemma Green
    Gail Green
    >>> print_names(["Carter",["Cathy"],"Downer",["Don"]])
    Cathy Carter
    Don Downer
    """

if __name__=="__main__":
    import doctest
    doctest.testmod(verbose=True)
```

6. A list contains pairs of items. The first of every pair is a string representing a surname / family name. The second item is a list containing one or more first names. Write a function which takes a list of this description as a parameter, and **returns** a list of full names. Here are some doctests:

```
def return_names(name_list):
    """
    >>> return_names(["Brown",["Bob","Betty"],"Green",["Geoff","Gemma","Gail"]])
    ['Bob Brown', 'Betty Brown', 'Geoff Green', 'Gemma Green', 'Gail Green']
    >>> return_names(["Carter",["Cathy"],"Downer",["Don"]])
    ['Cathy Carter', 'Don Downer']
    """

if __name__=="__main__":
    import doctest
    doctest.testmod(verbose=True)
```

7. Write a function `myreplace(s, old, new)` that returns a string with all occurrences of `old` replaced with `new` in the string `s`. Assume that `old` has only one character:

```
def myreplace(s, old, new):
   """
   >>> myreplace("Mississippi", "i", "I")
   'MIssIssIppi'
   >>> s = "I love spom!  Spom is my favorite food.  Spom, spom, spom, yum!"
   >>> myreplace(s, "o", "a")
   'I lave spam!  Spam is my favarite faad.  Spam, spam, spam, yum!'
   """
```

Your solution should pass the doctests above, but don't use the existing string replace method (make use of loops instead).

8. Redo the previous exercise, but this time assume that `old` has two characters.

```
def myreplace(s, old, new):
    """
    >>> myreplace("Mississippi", "is", "I")
    'MIsIsippI'
    >>> s = "I love spom!  Spom is my favorite food.  Spom, spom, spom, yum!"
    >>> myreplace(s, "om", "am")
    'I love spam!  Spam is my favorite food.  Spam, spam, spam, yum!'
    """
```

Your solution should pass the doctests above (again use loops). Instead of trying to solve the whole problem, use incremental development with the following intermediate goals:

(a) loop through `s` and print out all substrings of size 2.

(b) loop through `s` and print all substrings that match `old`.

(c) loop through `s` and print all substrings of size 2, except replace those substrings that match `old` with the string `new`.

9. Redo the previous exercise, except this time make no assumptions about the length of `old`.

10. Develop a function which returns a 0-filled matrix of given size. The function should pass the following doctests:

```
def zeros(n,m):
    """
    Return an n by m matrix of zeros. Each element should be unique.
    >>> A = zeros(3,2)
    >>> A
    [[0, 0], [0, 0], [0, 0]]
    >>> A[0][1] = 1
    >>> A
    [[0, 1], [0, 0], [0, 0]]
    """
```

11. *Extension Exercise:* Create a function called `insert_sorted` that inserts a new element into an already sorted list in order:

```
def insert_sorted(lst, element):
    '''
    >>> insert_sorted([1,3,5], 4)
    [1, 3, 4, 5]
    >>> insert_sorted([1,2,3], 0)
    [0, 1, 2, 3]
    >>> insert_sorted([1,2,3],4)
    [1, 2, 3, 4]
    '''
```

12. *Extension Exercise* Matrix multiplication. If you don't know how to do matrix multiplication, spend some time learning how to do it (the web is a good place to start - try http://www.intmath.com/matrices-determinants/4-multiplying-matrices.php or http://www.youtube.com/watch?v=qiLjdujcI2w&feature=related). Complete the body of the `matrix_multiply_2x2` function below, so that it passes the doctests.

```
def matrix_multiply_2x2(amatrix, bmatrix):
    """
    Return a matrix which is the result of multiplying amatrix by bmatrix.
    Assume that both matrices are of size 2x2. The resultant matrix should
    also be 2x2.

    >>> matrix_multiply_2x2([[1, 0], [0, 1]], [[2, 3], [4, 5]])
    [[2, 3], [4, 5]]
    >>> matrix_multiply_2x2([[2,3],[4,5]], [[1,0],[0,1]])
    [[2, 3], [4, 5]]
    >>> matrix_multiply_2x2([[1,2],[3,4]], [[5,6],[7,8]])
    [[19, 22], [43, 50]]
    """
```

13. *Extension Exercise* Extend your solution to exercise 12 so that `matrix_multiply` can accept matrices of any size (assuming they are compatible). You will need three nested loops.

```
def matrix_multiply(amatrix, bmatrix):
    """
    Return a matrix which is the result of multiplying amatrix and bmatrix.
    Assume that amatrix is an NxM matrix and bmatrix is an MxP matrix
    (this means amatrix has N rows and M columns, while bmatrix has M
    rows and P columns). The resulting matrix will be an NxP matrix.
    A matrix is represented as a list of lists as shown below.

                [1 4]
    The matrix  [2 5]  is represented by this list  [[1,4],[2,5],[3,6]]
                [3 6]

    >>> matrix_multiply([[1,2]],[[3],[4]])
    [[11]]
    >>> matrix_multiply([[1],[2]], [[3,4]])
    [[3, 4], [6, 8]]
    >>> matrix_multiply([[1,2,3],[4,5,6]], [[1,4],[2,5],[3,6]])
    [[14, 32], [32, 77]]
    >>> matrix_multiply([[1,2,3],[4,5,6]], [[1,4,7,10],[2,5,8,11],[3,6,9,12]])
    [[14, 32, 50, 68], [32, 77, 122, 167]]
    """
```

14. *Extension Exercise:* Dealing with dates is a common problem with web-form processing (how many times have you had to enter the date in a web form?). Read a date in from the user and convert it to the standard date format: DD/MM/YYYY.

    Users are likely to enter the date using several formats: E.g.

```
12/6/07
13-2-09
13 May 1996
```

This is quite a difficult problem. Start off with some of the simpler conversions, then add more as you go. Think about what you should do if the user enters an invalid date.

## 13.12 Mastery Level Complex - Practice Questions

Mastery Level Complex was a mastery level in the old COMP150 paper. It is not a mastery level in COMP151. However, these practice questions do test your mastery of the material covered in lessons 11-14, and this stuff will be very useful for both Mastery Level 7, Mastery Level 8, and Practical Test 2.

You will need to download support files - these are available on Blackboard. It is important that you do not alter them if you want the doctests to pass!

You should check your solutions with a demonstrator.

1. Write a function that takes the name of a file as a parameter.

   The file will contain several lines of ints which are separated by commas.

   The function should open the provided file and:
   - add up the even numbers on each line to produce a line total
   - concatenate the line total to a string, followed by a space
   - return the string after the whole file has been processed.

   Given a file containing:

   1,2,3,4
   2,4,6
   1,3,5,11,33,13,15,17

   Your function should return:

   '6 12 0 '

   Remember to have all the required files in the same directory as the python file.

   The function should pass the following doctests:

```
    def sum_evens(file_name):
    '''
    >>> sum_evens('numbers_1.txt')
    '164 12 0 '
    >>> sum_evens('numbers_2.txt')
    '7654 5360 1569200 46574 7635754 0 9859896 '
    '''
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

2. Write a function that takes 2 parameters, the name of a file, and an int.

   The file will contain several lines consisting of words separated by commas.

   The function should print the following for each line of text in the file:

   "Line " followed by the appropriate number, starting at 1

   Each word in the line with a length that is greater than the int parameter, followed by a space

   Note: If you use `print`, a `\n` character is written at the end, unless the print function uses the `end` argument to change that.

Remember to have all the required files in the same directory as the python file.

Your function should pass the doctests below:

```
def print_words_longer_than(file_name, x):
    '''
    >>> print_words_longer_than('words_1.txt', 5)
    Line 1: apricot nectarine rockmelon
    Line 2: parsnip turnip pumpkin carrot
    Line 3: oregano rosemary
    >>> print_words_longer_than('words_2.txt', 7)
    Line 1: elephant
    Line 2: crocodile alligator
    Line 3: blackbird
    '''
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

3. Write a function that returns a seating plan (in the form of a string) which is, thankfully, always rectangular. Rows are represented by capital letters and seats by numbers (starting from 1). Each seating position is followed by a space. The function should take, as its parameters, the number of rows required and the number of seats required in each row. The output of the function called with the arguments 3 and 4 should, when printed, look as follows:

```
A1 A2 A3 A4
B1 B2 B3 B4
C1 C2 C3 C4
```

You may assume(for now) that the number of rows will not exceed 26

Avoid using long string literals containing letters - that is no fun!

Remember:
ord('A') gives the ascii number of 'A' which is 65
chr(65) gives the char 'A'
Capital letters are contiguous.

Remember to have all the required files in the same directory as the python file.
Your function should pass the doctests below:

```
def seating_plan_simple(num_rows, num_seats):
    '''
    >>> seating_plan_simple(4,6)== open('school_play_simple.txt').read()
    True
    >>> seating_plan_simple(15,15)== open('opera_simple.txt').read()
    True
    '''

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

4. As for the question above but to save confusion, the letters O and I are not used since they look too much like zero and one. The number 13 is not used since superstitious people refuse to sit there. For an easy life you may assume that there will be no more than 24 rows.

   Your function should pass the doctests below:

```
def seating_plan_opt1(num_rows, num_seats):
    '''
    >>> seating_plan_opt1(4, 6)== open('school_play_simple.txt').read()
    True
    >>> seating_plan_opt1(15,15)== open('school_play_opt1.txt').read()
    True
    >>> seating_plan_opt1(15, 15)== open('opera_opt1.txt').read()
    True
    '''

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

5. Octal number conversions

   In computer science we sometimes need to work with numbers that are not decimal (base 10).

   To convert a positive decimal number, x, to a number in another base:

   1. Find the remainder when you divide x by the base (remember the mod function)
   2. Divide x by the base (using integer division)
   3. Repeat 1 and 2 until x is zero

   The answer (x in the new base) will be the remainders in reverse order. The answer should be returned as a string.

   E.g To convert 103 to base 7:

   | | |
   |---|---|
   | The remainder when we mod 103 by 7 is 5 | 5 will be the last digit of our answer |
   | 103 divided by 7 is 14 (x is now 14) | |
   | The remainder when 14 is divided by 7 is 0 | 0 will be the 2nd-last digit of our answer |
   | 14 divided by 7 is 2 (x is now 2) | |
   | The remainder when 2 is divided by 7 is 2 | 2 will be the first digit of our answer |
   | 2 divided by 7 is 0 (x is now zero) | |
   | At this point the loop stops and the answer is '205' | |

   You are required to write a function that takes a string of positive decimal integers, separated by spaces, and returns a string of the same values but converted to octal(base 8)

   Note:

   Your code must not use python's oct function.

   A decimal zero input should produce '0' in any base.

   You may find it easier to encapsulate, using another function that takes one decimal int and simply returns an octal string for that number.

Your function should pass the following doctests:

```
def to_oct_nums(num_string):
    '''
    >>> to_oct_nums('100 23 44 32 6 17 ')
    '144 27 54 40 6 21 '
    >>> to_oct_nums('0 1 8 64 512 4096 ')
    '0 1 10 100 1000 10000 '
    '''

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose = True)
```

## Get a demonstrator to review your work before you leave the lab

Once your work has been reviewed, you may progress to the next lesson or **Sign Out**.
Please do **not Shutdown** the lab computers.

# Lesson 14

# Number Representations

In this lesson, we are going to focus on number representations, but we will also touch on representing other things in the computer such as characters and emojis. First, though, we need to talk about what a decimal number represents.

## 14.1 Decimal, Binary, Hexadecimal and Octal

### 14.1.1 Decimal numbers

Everyone is familiar with the usual decimal numbers. Decimal numbers are based on the Hindu-Arabic number system, first introduced in India between the 1st and 4th centuries AD, adopted by Arabic mathematicians in the 9th century, and from there spreading to Europe in the 11th century or later. As an aside, Arabic mathematicians were the best in the world at around the 9th century, and this is where the words algebra[1] and algorithm[2] originate from.

Consider a number such as 342 - what exactly does it represent? It relies on two components - a base and exponentiation. For decimal numbers, the base is 10[3], and we have 10 individual digits from 0 to 9. Exponentiation comes into it based on the position of the digits in the overall number. Essentially, 342 means the following:

$$342 = 3 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

Each digit we add on the left adds 1 to the exponent. The scheme even works with real numbers, where each digit to the right means subtracting 1 from the exponent. For example, 342.789, represents the following:

$$342.789 = 3 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2} + 9 \times 10^{-3}$$

### 14.1.2 Binary numbers

Now that we have an understanding of decimal numbers, you might notice that there is no need for the base number to be 10. In fact, we can make that number be any number we like as long as it's bigger than 1. For example, here's a binary number:

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

[1] al jabr in Arabic

[2] from the mathematician Muhammad ibn Musa al-Khwarizmi. al-Khwarizmi was westernised to algorithm.

[3] I realise this definition is circular.

and a binary number with a decimal point:

$$1.011 = 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

### 14.1.3 Octal and Hexadecimal

Computers run on binary numbers, so many important numbers to do with computers are powers of two. For example, a byte is 8 bits, a kilobyte is $2^{10} = 1024$ bytes, a megabyte is $2^{20}$ bytes.[4] Although computers run on binary, it's not a very convenient representation to communicate in, because it's quite lengthy. For example, memory locations[5] in the computer are addressed via binary numbers, but Python programs normally do not report this location in binary, instead it often uses hexadecimal. Hexadecimal uses base 16 which is easy to convert to and from binary, because each character in hexadecimal is associated with 4 digits in binary and vice versa. If you look at the variables tab in Thonny, you will sometimes see something like `<function print_parity at 0x10c04e320>`. That last bit is a hexadecimal number reporting where in memory the function `print_parity` resides. The $0x$ at the start is a signifier specifying that the number is hexadecimal. Now, with base 16, there's a problem - there aren't enough digits to go up to 15, so we use letters instead $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$. The character $a$ representing the number 10, $b$ representing 11 etc. So:

$$0xe320 = 14 \times 16^3 + 3 \times 16^2 + 2 \times 16^1 + 0 \times 16^0 = 58144$$

Actually, in Python hexadecimal is a valid integer representation:

```
>>> print(0xe320)
58144
```

Octal is somewhat less prevalent, but still used sometimes and is base 8, and uses the prefix $0o$. Similarly, binary numbers can be specified using the prefix $0b$:

```
>>> print(0o7777)
4095
>>> print(0b1111)
15
```

To convert the other way, that is from integer to binary or octal or hexadecimal, we can use the corresponding conversion functions:

```
>>> bin(63)
'0b111111'
>>> hex(63)
'0x3f'
>>> oct(63)
'0o77'
```

But note that the result of these functions is a string and not a number.

## 14.2 ASCII and Unicode

All things in a computer must be represented by a binary number – binary numbers are the fundamental object that a computer manipulates. That also includes strings and the characters that make up strings.

[4] Actually, there's a bit of confusion around the terms kilobyte and megabyte etc, SI defines kilo as 1000 and mega as 1000000, but in computing it commonly means the power of two version.

[5] In Random Access Memory or RAM

Since what's in the computer are binary numbers, when strings are printed to the screen, there needs to be a translation from the number used to represent a character to the visual representation of the character. A common translation that has been used by many western computer systems is the ASCII[6] character set. However, ASCII is an 8-bit format, which means it can only store 256[7] different characters. This is a problem because different languages use different character sets (think of the 10000 or so Chinese pictograms). Nevertheless, ASCII is still commonly used (and is part of Unicode anyway), and Python can convert from a character to its ASCII number using the function `ord` and from a number to a character using the function `chr`:

```
>>> ord('a')
97
>>> ord('A')
65
>>> chr(97)
'a'
>>> chr(65)
'A'
```

You might remember from Chapter 8 that all the upper case letters come before the lower case letters when doing comparisons — the ASCII encoding is the reason why. `ord` and `chr` also work with more general Unicode characters which are described below.

To address the problems with various different encodings, Unicode was introduced and is now the standard representation used in all modern computer systems. Unicode is able to represent over 1 million different characters, and this includes the ASCII characters, characters from all the world's languages, and other sorts of glyphs[8] such as emojis. There are two explicit forms of Unicode in Python — UTF-16 and UTF-32. UTF-16 characters are specified using a `\u` escape character followed by four hexadecimal digits (so 16 bits in total), and UTF-32 characters are specified using a `\U` escape character followed by 8 hexadecimal digits (32 bits). Unicode characters don't work especially well with the documentation system used to create this book (LaTeX), but you can play around with printing out characters and emojis using Thonny. Try out some of the following:

```
# a chess knight
print('\u265E')
# smiley face
print('\U0001F600')
# poop emoji
print('\U0001F4A9')
```

Note that UTF-16 strings must have exactly 4 characters after the escape, and UTF-32 strings must have 8 characters. If you look at the emoji chart at

http://www.unicode.org/emoji/charts/full-emoji-list.html

and want to print some of them out, you will need to zero pad them on the left.

## 14.3 Integers

We know how to specify binary numbers and that makes it easy to represent positive integers, but what about negative numbers? The simplest way of representing negative numbers is to include a sign bit on

[6] American Standard Code for Information Interchange
[7] $256 = 2^8$
[8] A glyph is the pictorial representation of the character.

the far left of the number — a 0 in the sign bit indicates a positive number, and a 1 indicates a negative number. But this representation has the problem of there being two zeros (positive zero and negative zero). Therefore python uses a representation called twos-complement. The details don't really matter for our purposes, but if you're interested you can have a look at

https://en.wikipedia.org/wiki/Two%27s_complement.

Most integers are represented in Python using 64 bits, but that detail is hidden away from the Python programmer. Integers can be of any size, and if they require more than 64 bits, then a larger representation is used automatically. This means you, as a programmer, don't really need to worry about integer representations in Python. This isn't true in all languages. Many languages use 64[9] bits as the representation size for integers, and if you add two large integers, this can cause overflow (the result can't be represented). So if you are using a different language for some programming, be mindful that overflow can occur when dealing with integers.

## 14.4 Floating point

Which brings us to the real topic of this lesson - floating point numbers. Before delving into floating point, let's consider an alternative representation for real numbers: fixed point representation. In such a representation, we simply fix where the decimal point is in all numbers. For example, let's say we decide to have 8 bit fixed-point numbers where the left-most 4 bits are before the decimal point, and the right-most 4 bits are after the decimal point (we're going to ignore negative numbers). For example:

$$1011.1000 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} = 11.5$$

The problem with this representation is that we are wasting those 3 bits on the right that are all 0. This particular representation means the maximum number I can represent is $15$ plus a bit. Conversely, the best precision I can hope for is $2^{-4} = 0.0625$. The representation is simple, but quite limiting.

Now let's consider scientific notation for real numbers. For example:

$$\begin{aligned} 342.56 &= 3.4256 \times 10^2 \\ 0.00000000034256 &= 3.4256 \times 10^{-10} \\ 342560000 &= 3.4256 \times 10^8 \end{aligned}$$

What we notice about all of these numbers in scientific notation is that the number of significant digits is the same regardless of the magnitude of the number, and we don't waste any space or precision representing unneeded zeros. This is the basis of floating point representations, and the only thing to do is decide how many bits for each part of the number. In particular, a floating point number is represented as:

$$\text{significand} \times 2^{\text{exponent}}$$

where we only represent the significand and the exponent (plus a sign bit). In Python, all floats make use of 64 bits (not true in all languages). There is a sign bit, 52 bits for the significand and 11 bits for the exponent. Since the significand is stored in binary, the first bit is always a 1 (the part to the left of the decimal), and therefore we don't need to store it. This means that the 52 bits of storage give us 53 bits of precision. Also, the exponent can be positive or negative. Rather than specify a sign bit, the exponent has a built in bias. All 0s and all 1s in the exponent are reserved for special numbers like 0 and infinity, the number 00000000001 represents the number -1022, and 11111111110 represents 1023, with the binary strings in between representing the relevant numbers between -1022 to 1023. In this way, the exponent ranges from -1022 to 1023. This looks like the smallest magnitude number bigger than zero would be $2^{-1022}$, but actually a double can go down to $2^{-1074}$, because in these cases, the significand is allowed to have leading zeros after

[9] Earlier computers used 16 bits or 32 bits, but most now use 64 bits.

the decimal. In any case, the consequence of this representation is we can represent a non-zero number as small as $2^{-1074}$ and a non-infinite number as big as $2^{1023}$, which is an enormous range of magnitudes. $2^{-1074}$ has 324 zeros after the decimal point, so it is tiny, whereas $2^{1023}$ is $8.988466 \times 10^{307}$, so it is a very large number indeed.

But a binary number with 64 bits can only represent $2^{64}$ distinct values. How can floating point numbers possibly represent all the numbers between $2^{-1074}$ and $2^{1023}$. The answer is that floating point numbers can't represent all those numbers. What we've gained in terms of flexibility, we've given up in terms of a dense representation of numbers. In fact, the number of real numbers between any two numbers is known to be uncountably infinite, and even the number of rational numbers between any two numbers is countably infinite[10], so there is no way to represent them on the computer. Most of the time this doesn't matter, and we can compute things as if a floating point number was a real number, but sometimes it really does matter, and we need to take extra care. Taking extra care is something that could fill up an entire other course[11], and we're not going to do that, but I am going to mention two issues that can occur, and that you might come across.

### 14.4.1 Some numbers can't be represented exactly

There are two main ways that numbers can't be represented exactly, even if they are within the allowable range of floats. The first way is because the number is irrational, like $\pi$, and has an infinite number of non-zero digits after the decimal point. But there is a more surprising way that numbers can't be represented exactly. Because most floats use powers of 2 in the exponent, even some simple numbers require an infinite number of non-zero digits after the decimal point. Consider the decimal number $0.1$. This number is $1 \times 10^{-1}$. Let's see how we can represent that as a power of 2. Well, $2^{-3} = 0.125$, so that's too big, and $2^{-4} = 0.0625$, so that's too small, so the binary representation will look like $1.xxxx \times 2^{-4}$, where the x's are either 0 or 1. Let's try a few numbers by trial and error:

$$
\begin{aligned}
1.1 \times 2^{-4} &= 1 \times 2^{-4} + 1 \times 2^{-5} && = 0.09375 && \text{\# too small} \\
1.11 \times 2^{-5} &= 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} && = 0.109375 && \text{\# too big} \\
1.101 \times 2^{-5} &= 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-7} && = 0.1015625 && \text{\# still too big} \\
1.1001 \times 2^{-5} &= 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-8} && = 0.09765625 && \text{\# too small} \\
1.10011 \times 2^{-5} &= 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-8} + 2 \times 2^{-9} && = 0.099609375 && \text{\# still too small}
\end{aligned}
$$

In fact, we can keep going like this forever: $1.10011001100110011$ and while we keep getting closer and closer, we never quite reach 0.1. So if we try to print out $0.1$ with many digits of precision, we get:

```
>>> print('{:.24e}'.format(0.1))
1.000000000000000055511151e-01
```

[10] Uncountably infinite is a bigger number than countably infinite!
[11] Such a course is usually called Numerical Analysis or similar.

### 14.4.2 Sometimes precision is lost

The order in which computations are made can sometimes affect the outcome. Not very often, but sometimes. In particular, adding a lot of small numbers to a big number may not have the effect you might think. For example, consider the following snippet of code:

```
v1 = 1e9
v2 = 1e-6
for i in range(int(1e6)):
    v1 = v1 + v2
print('answer is', v1 - 1e9)
```

which produces the output:

```
answer is 0.95367431640625
```

Clearly, the answer should be $1.0$ since we're adding $10^{-6}$ 1000000 times, and $1000000 \times 10^{-6} = 1.0$. But we're doing the computation by adding a small number many times to a big number, and since not all numbers can be represented, sometimes during the computation, the intermediate result is not quite accurate, and those inaccuracies add up over time. Even though the individual inaccuracies are very small, cumulatively, the resultant inaccuracy is quite large. Generally, one needs to be careful when adding many small numbers to a large number.

## 14.5 Exercises

1. Binary to decimal. Convert the following binary numbers to base 10 by hand (don't use the computer):
   (a) 10
   (b) 101
   (c) 1000
   (d) 1111

2. Binary to hexadecimal. Convert the following binary numbers to base 16 by hand. Hint: each group of 4 digits makes up 1 hex character.
   (a) 1000
   (b) 11111000
   (c) 10000101
   (d) 01010010

3. Write a function that takes as input a string in either binary, octal or hexadecimal format and returns the decimal integer representation of that number. The function should check the first two characters of the string to determine what input type it is. To actually do the conversion, for example from binary, you need to tell the `int` conversion routine what the base is. For example, `int('0b11',2)` returns 3, but `int('0b11')` raises an error. Your function should pass the following doctests:

```
    def convert_to_int(str_rep):
        >>> convert_to_int('0b11')
        3
        >>> convert_to_int('0xfae')
        4014
        >>> convert_to_int('0o545')
        357
```

4. From Blackboard, download the movie ratings csv file. Use a csv reader (see Chapter 9 for a refresher) to read in each row. For each movie, print out the movie name, the movie rating, and if the rating is below 2, print out the poop emoji (it stinks), if it's between 2 and 3.5, then print a neutral face emoji, and if it's above 3.5, a smiley face. You can find the Unicode codes at http://www.unicode.org/emoji/charts/full-emoji-list.html.

5. Rewrite the following code, so that it produces the right answer:

```
v1 = 1e9
v2 = 1e-6
for i in range(int(1e6)):
    v1 = v1 + v2
print('answer is', v1 - 1e9)
```

   Hint: adding a small number to a small number does not cause imprecision.

# Lesson 15

# Introduction to NumPy - arrays

For scientific applications in Python, the foundational library is called NumPy which is short for numerical Python. NumPy forms the basis for a large number of discipline specific computing tools that include: quantum computing, statistics, signal processing, astronomy, psychology, bioinformatics, chemistry, and more. We're not going to cover all those things, but we may sneak in a couple of examples over the rest of the semester. You might wonder why there is a library like NumPy. NumPy essentially provides a foundational data structure (the NumPy array) and a bunch of algorithms for manipulating arrays. An array is very similar to a list (or multidimensional list), and is used to represent vectors, matrices and higher order objects like tensors. But why do we need an array if we've already got a list? After all, we can certainly represent vectors and matrices with lists (see Section 13.8). It comes down to efficiency. Python is a great language and environment, but because it is interpreted, it can be slow compared to compiled languages like C or C++. But Python has a neat trick - libraries can be written in C or C++ and imported into Python. Doing so, we get the advantages of programming in Python, with the speed of a compiled language. That's what NumPy offers - very fast operations wrapped up in the convenience of Python.

## 15.1 Installing NumPy

NumPy is not a standard library so is not installed by default - you will need to install it before you are able to use it. Thankfully Thonny makes the process of installing additional libraries painless. Go to "Tools $\rightarrow$ Manage Packages", search for numpy, and then choose "Install". You should initially see a window that looks like the window shown in Figure 15.1. After you install numpy, you will be able to import it and use all of its data structures and functions.

## 15.2 NumPy Arrays

A NumPy array is like a Python list (or nested list) except for two very important distinctions: all elements of the array must be of the same type (quite often a float); and the shape of the array must be regular (for example, each row in a 2 dimensional array must have the same length). This means that arrays are regular grid-like structures of a fixed number of dimensions. One-dimensional (1d) arrays are similar to a single list. Two-dimensional (2d) arrays are like a nested list. Three-dimensional arrays are like a list of nested arrays, etc. Here is an example bit of code creating several different arrays:

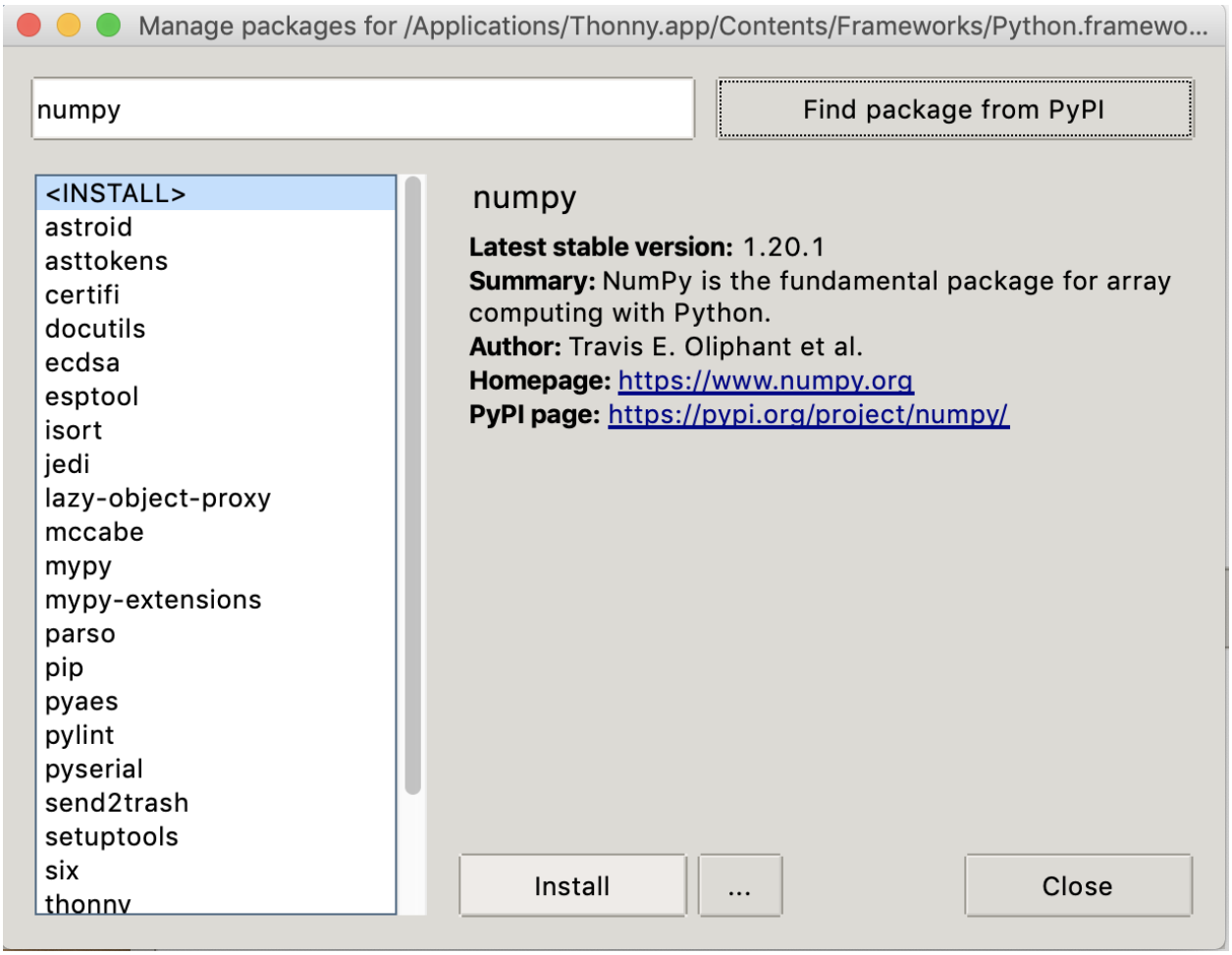



Figure 15.1: The Package Manager

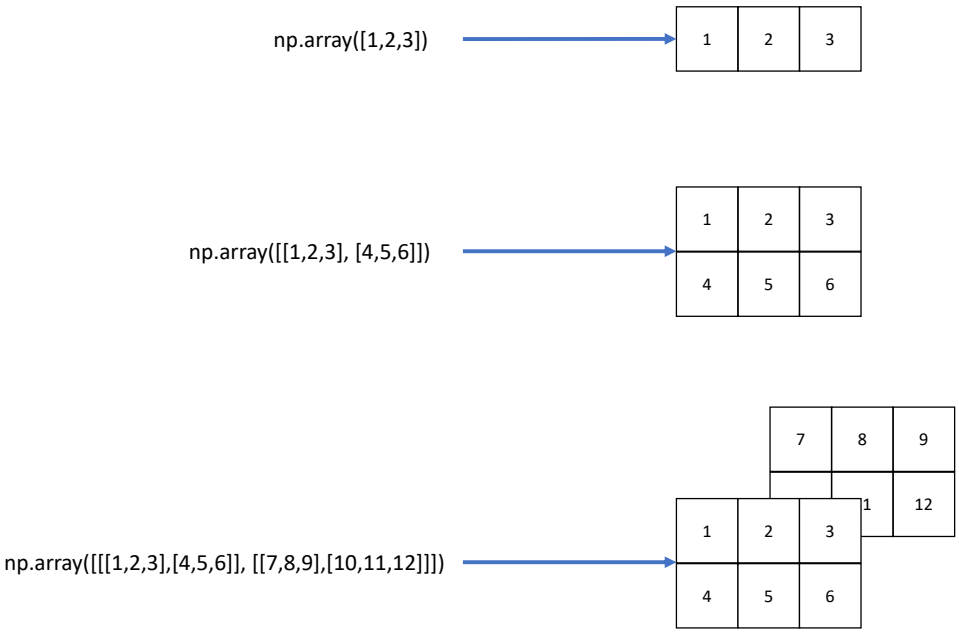



Figure 15.2: A 1D, 2D and 3D array.

```
import numpy as np

a1d = np.array([1,2,3])
a2d = np.array([[1,2,3],[4,5,6]])
a3d = np.array([[[1,2,3],[4,5,6]], [[7,8,9],[10,11,12]]])
```

Two remarks on the code are in order. First, although the module is `numpy`, it is almost universal to import it as `np`. Second, it is quite common to create arrays from lists, although there are many other ways of creating arrays as we will see later. A visual view of what the arrays look like[1] is shown in Figure 15.2.

## 15.3 Shape of an array

The shape of an array indicates the number of dimensions and the size of each dimension. This is stored in the array attribute `shape` which is just a tuple[2] of numbers. The shapes of the above arrays are as follows:

```
>>> print(a1d)
[1 2 3]
>>> a1d.shape
(3,)
>>> print(a2d)
[[1 2 3]
 [4 5 6]]
```

[1] At least metaphorically. The actual phsyical layout of the arrays in memory is somewhat more complicated, but not that interesting for our purposes.

[2] A tuple is like a list, but not quite the same.

```
>>> a2d.shape
(2, 3)
>>> print(a3d)
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
>>> a3d.shape
(2, 2, 3)
```

## 15.4 Indexing and Slicing

Array elements can be accessed in the same way as list elements – using indexing. The number of indices to use is usually the same as the length of the array shape, but it can be fewer, as we will see below. As with lists, the elements of an array are 0-indexed:

```
>>> a1d[0]
1
>>> a2d[0,0]
1
>>> a3d[0,0,0]
1
```

As with lists, parts of an array can be extracted using slicing:

```
>>> a3d[:,:,0:2]
array([[[ 1,  2],
        [ 4,  5]],

       [[ 7,  8],
        [10, 11]]])
>>> a3d[:,:,0:1]
array([[[ 1],
        [ 4]],

       [[ 7],
        [10]]])
```

Where ":" by itself, means use that whole dimension. If fewer indices than the number of dimensions are used, then ":"s are assumed for the latter dimensions. For example:

```
>>> a3d[0,1]
array([4, 5, 6])
```

is equivalent to:

```
>>> a3d[0,1,:]
    array([4, 5, 6])
```

## 15.5 Reshaping arrays

An array can be reshaped to any shape with the same total number of elements. The easiest reshaping is the `flatten` operation which converts any dimensional array into a 1 dimensional array:

```
>>> a3d.flatten()
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

But other reshapes are possible using the `np.reshape` function[3]:

```
>>> np.reshape(a3d, (2,6))
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

The `reshape` function does not alter the underlying data and also doesn't copy the data, it just provides a new view into the data via aliasing[4]. This means that changes to a reshaped array will be reflected in the original data:

```
>>> newa = np.reshape(a3d, (2,6))
>>> newa[0,0] = 72
>>> print(newa)
[[72  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
 >>> print(a3d)
 [[[72  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]
```

The reason for this behaviour has to do with efficiency. Numpy arrays can become very large, and copying this data can become extremely expensive in both time and space. It is usually better to leave the decision to copy data to the programmer. In this case, data can be copied using `np.copy`. Note that unlike lists, slicing a numpy array does not copy the underlying data (again for efficiency reasons).

## 15.6 Array Concatenation

List concatenation is easy using the + operator, and the semantics[5] of list concatenation is also straightforward – the two sets of elements of the list get concatenated to form a longer list. It's not quite so straightforward for concatenating arrays, because all the elements of an array must be of the same type. For example, the elements of a 3D array are actually 2D arrays. Also, multidimensional arrays have several dimensions over which concatenation could occur. The basic rule is that the dimensionality of arrays being concatenated must be identical *except* in the dimension of concatenation. In any case, `np.concatenate` is the essential method. Here are a few examples and their results:

[3] There is an equivalent reshape method: `a3d.reshape((2,6))`

[4] If possible. Sometimes this isn't possible and a copy of the data is made. But such cases are a little unpredictable, and it's best to assume that a view of the original data will be returned.

[5] the meaning or result of an operation

```
>>> np.concatenate((a1d,a1d[0:1]))
array([1, 2, 3, 1])
>>> np.concatenate((a1d,[7,8]))
array([1, 2, 3, 7, 8])
>>> np.concatenate((a2d,a2d))
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
>>> np.concatenate((a2d,a2d),axis=1)
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
>>> np.concatenate((a2d,a3d[0]),axis=1)
array([[ 1,  2,  3, 1,  2,  3],
       [ 4,  5,  6, 4,  5,  6]])
>>> np.concatenate((a2d,a3d[1]),axis=0)
>>> np.concatenate((a2d,a3d[1]),axis=0)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

The `axis` parameter indicates along which dimension to do the concatenation. In the case of 2D arrays, `axis=1` means adding columns and `axis=0` means adding rows. It's pretty hard to remember all this stuff, so I usually just experiment with a bunch of examples when I need to figure out how to do something. If you want to see what's happening, I suggest you try out a bunch of your own examples.

## 15.7 Basic array operations and broadcasting

The semantics of numpy array operations can be quite complicated. In general, the semantics are a superset of the operations allowed in linear algebra (vectors, matrices, tensors). The extra allowable operations make some things very convenient, but also make some of those things difficult to understand. Again, experimenting with a bunch of examples is a good way to get an understanding of what is going on. We're going to start with two of the simplest operations: addition and multiplication by a scalar.

Adding two arrays is straightforward if the arrays have the same number of dimensions. The result is simply the sum of each of the corresponding elements:

```
>>> a1d+a1d
array([2, 4, 6])
>>> a2d+a2d
array([[ 2,  4,  6],
       [ 8, 10, 12]])
>>> a3d+a3d
array([[[150,   4,   6],
        [  8,  10,  12]],
        [[ 14,  16,  18],
         [ 20,  22,  24]]])
```

But we can also add a scalar[6] to an array. In this case, the number is added to every element in the array:

[6] a single number

```
>>> a1d+5
array([6, 7, 8])
>>> a2d+1000
>>> a2d+1000
array([[1001, 1002, 1003],
       [1004, 1005, 1006]])
```

Which can be quite convenient. Note that this is not an allowed operation in linear algebra - it is an example of what is called **broadcasting**. In this case the operation of addition by a scalar is broadcast to all elements of the array.

Multiplication by a scalar has the same sort of effect as addition by a scalar (and is a valid linear algebra operation) - each element in the array is multiplied by the same scalar:

```
>>> a2d*3
array([[ 3,  6,  9],
       [12, 15, 18]])
```

We can then start building more complicated expressions:

```
>>> a1d*3+a1d*2
array([ 5, 10, 15])
```

Multiplication also works with arrays of the same size, but this definitely *does not* follow the rules of linear algebra. Again, the elements are multiplied component-wise:

```
>>> a1d*a1d
array([1, 4, 9])
```

Both addition and multiplication by a scalar are the simplest sort of broadcasting possible. But broadcasting also works for any dimensional arrays[7] as long as the appropriate conditions are met. Those conditions depend on the shape of the two arrays upon which the operation is being performed. Generally, two arrays are compatible for broadcasting an operation if, for each dimension:

1. the size of the dimensions are equal; or
2. the size of one of the dimensions is 1.

The size of the resulting array essentially stretches all the size 1 dimensions to match the larger dimension of the two. Don't worry about it too much - basically no-one remembers this stuff, and I personally only use very simple examples of broadcasting most of the time. Here are some simple examples with a bit of explanation:

```
>>> a2d+a1d
array([[2, 4, 6],
       [5, 7, 9]])
```

In this case, `a2d` has shape $(2, 3)$ and `a1d` has shape $(3, )$. For broadcasting a 1D array with any other size, the last dimensions have to agree (in this case, both with size 3). It gets a bit more complicated with higher dimensional arrays:

[7] scalars can be thought of as 0-dimensional entities

```
>>> a3d+a1d
array([[[ 2,  4,  6],
        [ 5,  7,  9]],
       [[ 8, 10, 12],
        [11, 13, 15]]])
>>> a3d+a2d
array([[[ 2,  4,  6],
        [ 8, 10, 12]],
       [[ 8, 10, 12],
        [14, 16, 18]]])
```

In both of these cases, the trailing dimensions match, so broadcasting is allowed. Much more complicated broadcasting is possible, but that can make code difficult to understand, so unless there is a very good reason for it (e.g. efficiency), my advice is to stick with the simpler cases.

## 15.8 Glossary

**array** A regular grid-like data structure of any number of dimensions.

**broadcasting** How numpy operations over arrays of different sizes or dimensions are handled.

**semantics** The meaning or effect of an operation.

## 15.9 Laboratory Exercises

1. Create a one-dimensional array consisting of the numbers from 1 to 100 (hint: first use the range function).

2. Reshape the array from the previous question so that it is a two-dimensional array with shape (10,10).

3. Create a two-dimensional array of shape (10,10) that looks like the following:

```
array([[1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009],
       [1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019],
       [1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029],
       [1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039],
       [1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049],
       [1050, 1051, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059],
       [1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069],
       [1070, 1071, 1072, 1073, 1074, 1075, 1076, 1077, 1078, 1079],
       [1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089],
       [1090, 1091, 1092, 1093, 1094, 1095, 1096, 1097, 1098, 1099]])
```

4. Create a two-dimensional array of shape (10,10) that looks like the following:

```
array([[  0,   2,   4,   6,   8,  10,  12,  14,  16,  18],
       [ 10,  12,  14,  16,  18,  20,  22,  24,  26,  28],
       [ 20,  22,  24,  26,  28,  30,  32,  34,  36,  38],
       [ 30,  32,  34,  36,  38,  40,  42,  44,  46,  48],
       [ 40,  42,  44,  46,  48,  50,  52,  54,  56,  58],
       [ 50,  52,  54,  56,  58,  60,  62,  64,  66,  68],
       [ 60,  62,  64,  66,  68,  70,  72,  74,  76,  78],
       [ 70,  72,  74,  76,  78,  80,  82,  84,  86,  88],
       [ 80,  82,  84,  86,  88,  90,  92,  94,  96,  98],
       [ 90,  92,  94,  96,  98, 100, 102, 104, 106, 108]])
```

Hint: try adding a slice of the original one-dimensional array to the original two-dimensional array.

5. Create a two-dimensional array of shape (10,10) that looks like the following:

```
array([[   0,    1,    4,    9,   16,   25,   36,   49,   64,   81],
       [ 100,  121,  144,  169,  196,  225,  256,  289,  324,  361],
       [ 400,  441,  484,  529,  576,  625,  676,  729,  784,  841],
       [ 900,  961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521],
       [1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401],
       [2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481],
       [3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761],
       [4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241],
       [6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921],
       [8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]])
```

Hint: exponentiation works on arrays.

6. Given the following three-dimensional array:

```
a3d = np.array(range(90)).reshape((3,3,10))
```

which looks like:

```
array([[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]],

       [[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
        [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]],

       [[60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
        [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
        [80, 81, 82, 83, 84, 85, 86, 87, 88, 89]]])
```

Using slicing:

(a) extract out the sub-array containing the numbers 60, 61, 62, 63, 64, 65, 66, 67, 68, 69.

(b) extract out the sub-array containing the numbers 4, 14, 24, 34, 44, 54, 64, 74, 84. What shape is the resulting array?

(c) extract out the sub-array containing the numbers 0, 30, 60.

(d) extract out the sub-array containing the numbers 66, 76, 86.

# Lesson 16

# Some more NumPy

In this lesson, we are going to look at some ways of using numpy and in particular numpy arrays. We are also going to introduce a little bit of visualisation using the **matplotlib** library.

## 16.1 Mathematical functions and visualisation

Numpy is an example of vectorised code. This means the basic operations happen across all elements of an array implicitly rather than having to program such things explicitly, if we were using lists for example. Good examples of this include adding a scalar to each element in the array, or multiplying a whole array by a scalar as we saw in the previous lecture. This makes numpy excellent for visualising mathematical functions because we can easily evaluate a function across a range of inputs in a single statement.

The basic idea for doing so is to first create an array containing all the values at which you want to evaluate the function. You can think of these as values along the x-axis of a two-dimensional plot. The easiest way to generate these x-values is to use the **np.arange** function. `np.arange` is a bit like Python's `range` function, except it can generate floating point numbers too. It returns a one-dimensional numpy array and has the form:

```
    np.arange(START, STOP, STEP)
```

where START is the starting point, STOP is the stopping point, and STEP is how far to step each time. This is best explained with examples:

```
>>> np.arange(0,1,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> np.arange(0,10,0.5)
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
       6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
>>> np.arange(-1, 1, 0.2)
array([-1.00000000e+00, -8.00000000e-01, -6.00000000e-01, -4.00000000e-01,
       -2.00000000e-01, -2.22044605e-16,  2.00000000e-01,  4.00000000e-01,
        6.00000000e-01,  8.00000000e-01])
>>> np.arange(1,0,-0.1)
array([1. , 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1])
```

As with the range function, the starting point is included in the array, but not the end point (it finishes one before the end). Also you might notice strange things about the third example: first, the output is in

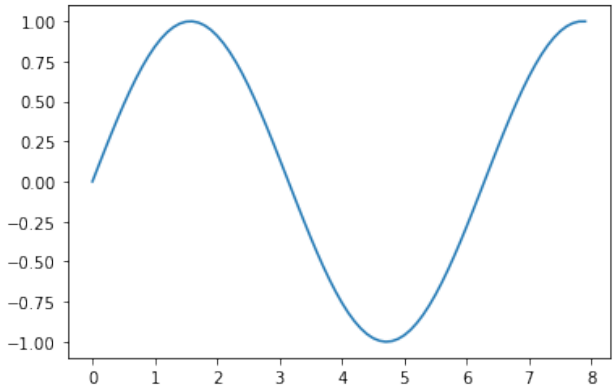



Figure 16.1: A visualisation of the sine function.

**scientific notation** which is of the form `a.be+c` which means $a.b \times 10^c$. For example, `2.00000000e-01` means $2.0 \times 10^{-1}$ which equals 0.2. The second strange thing is that the 6th entry is not zero even though we expect it to be. This is a quirk of floating point numbers - not all numbers can be represented exactly, and sometimes there is a little bit of error when we do calculations. We've covered this in Lesson 14. Note that `-2.22044605e-16` is $-2.22044605 \times 10^{-16} = 0.0000000000000000222044605$ which is a number very close to zero.

OK, so let's say we want to evaluate and visualise the sine function over the range from 0 to 8, how can we do that. First we need to generate an $x$ array containing all the $x$ values we want to evaluate sine at. Then we can simply call the `np.sin` function on that array:

```
x = np.arange(0,8.0,0.1)
y = np.sin(x)
```

Note that we have to use the `np.sin` function and not the `math.sin` function because the former is vectorised and will work on arrays, but the latter is not (and will raise an error).

The variable $y$ will now contain all the values of sine corresponding to the values in $x$. How do we visualise that? With **matplotlib** it's pretty straightforward:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(x,y)
>>> plt.show()
```

Which produces the plot shown in Figure 16.1.

Note that the resultant graph is just an approximation of the sine function. The function is not quite smooth, and is made up of a bunch of straight lines rather than smooth curves. We can see this more clearly if we sample the sine function at a coarser set of x values:

```
x = np.arange(0,8.0,0.5)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```

Which produces the plot shown in Figure 16.2. This is actually a key concept in almost all scientific computing. Almost all calculations and visualisations are approximations. This can sometimes lead to real and significant problems - sometimes small errors get magnified into very large errors. You should always keep

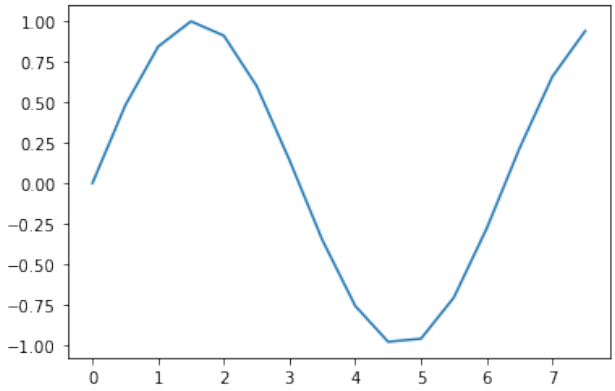



Figure 16.2: A coarse visualisation of the sine function.

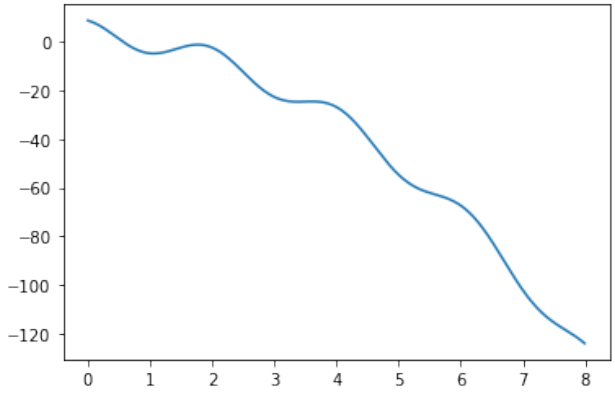



Figure 16.3: A more complicated function.

a sceptical eye on any results that come out of a computer and at the very least try to do some very rough mental checks to see if they are in the right ballpark.

Of course, we can use this same idea to evaluate and visualise very complicated functions quite easily. For example:

```
x = np.arange(0,8.0,0.01)
y = 3/(1+x**3) - 2*x**2 + 1.0/np.exp(x) + 5*np.sin(3*x+2)
plt.plot(x,y)
plt.show()
```

Which results in the plot in Figure 16.3

Numpy can also be very useful when dealing with measured data. For example, we might have measure the current flow in an alternating current circuit every 0.1s for 10s resulting in the following measurements (conveniently stored in an array):

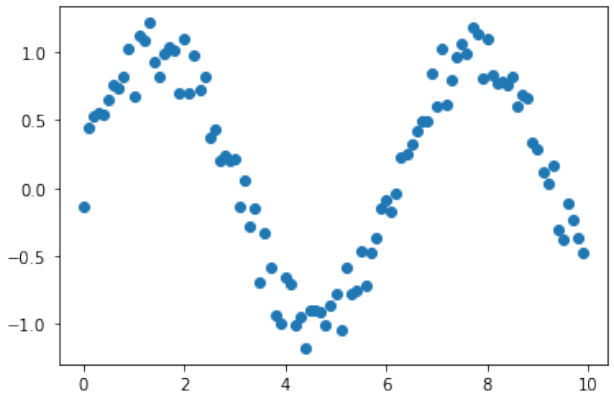



Figure 16.4: A bunch of measured data.

```
array([-0.12835682,  0.18596936,  0.23654713,  0.2796275 ,  0.59114026,
    0.39826842,  0.70068429,  0.88034996,  0.60976645,  1.04619449,
    1.13620898,  0.87262135,  1.04736508,  0.93876146,  1.21469805,
    0.89037785,  0.94068521,  0.94597761,  1.06122321,  1.09028531,
    0.80041781,  0.71343829,  0.51410037,  0.76877852,  0.73127998,
    0.66532006,  0.55006689,  0.25200979,  0.39701054,  0.21565686,
    0.26054313,  0.15308065, -0.01319132, -0.36022887, -0.4820507 ,
   -0.22954712, -0.54310944, -0.72031935, -0.44171886, -0.89947237,
   -0.74863037, -0.9143804 , -0.72209721, -1.13274576, -0.97663827,
   -0.95940134, -0.84512469, -1.18091908, -1.03191475, -0.95026465,
   -0.6820832 , -1.1065895 , -0.60041595, -0.94003499, -0.81783137,
   -0.53989467, -0.44793819, -0.58512245, -0.39113487, -0.31119101,
   -0.33379554,  0.03308104,  0.19503943,  0.07853923,  0.161786  ,
    0.13886514,  0.57320604,  0.71930823,  0.33155035,  0.74126252,
    0.82553916,  0.56728356,  1.05300633,  0.9956412 ,  0.74976175,
    1.02257962,  1.05046524,  0.90830053,  0.7868493 ,  1.01883985,
    0.83108819,  1.04035655,  1.07573499,  1.07675651,  0.94717483,
    0.5775733 ,  0.78433261,  0.69026678,  0.62098487,  0.20394453,
    0.46346074,  0.08737522,  0.10297696,  0.11356248, -0.16844309,
   -0.15598859, -0.21328038, -0.33740179, -0.31021406, -0.6550336 ])
```

We can visualise those points using a scatter plot shown in Figure 16.4 (assuming the array is assigned to a variable called `ymeas`):

```
x = np.arange(0,10,0.1)
plt.scatter(x,ymeas)
plt.show()
```

These data points look like they might be forming a noisy sine curve, and from what we know about alternating current, that is a reasonable hypothesis. What if we just plot the basic sine curve in addition to the scatter points (shown in Figure 16.5):

```
x = np.arange(0,10,0.1)
plt.scatter(x,ymeas)
plt.scatter(x,np.sin(x))
plt.show()
```

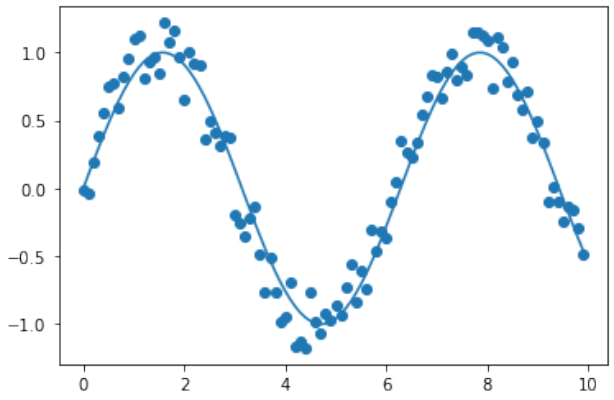



Figure 16.5: Measured data with a sine function.

That looks like a pretty good fit, but I wonder what the error is? A common measure of error of fit is the sum of squared errors which is computed as:

$$E = \sum_i (y_i - y_i')^2 \tag{16.1}$$

where $E$ is the error, $y_i$ is the value on the actual curve (in this case the sine function) and $y_i'$ is the value from the measured data. We can compute that error very easily with numpy and it looks a lot like the mathematical notation:

```
# we assume the measured data is already in the array ymeas
x = np.arange(0,10,0.1)
y = np.sin(x)
error = np.sum((y-ymeas)**2)
print(error)
```

Which results in an error of 2.860. But we can do a little bit better using the function $y = \sin(x + 0.1)$:

```
x = np.arange(0,10,0.1)
y = np.sin(x+0.1)
error = np.sum((y-ymeas)**2)
print(error)
```

Which results in an error of 2.184. However, you wouldn't be able to easily tell which was better by just looking at the graphs (shown in Figure 16.6), which highlights the importance of getting actual numerical results rather than just relying on visualisations. We will look into this topic a bit more when we look at some regression problems in another chapter.

## 16.2 Dot products and matrix multiplication

Fundamental to almost all data analysis and scientific computing applications is linear algebra. Linear algebra consists largely of the operation of matrices and vectors. A vector is similar to a list of numbers or a one-dimensional array, but it has a specific geometric interpretation. If you have done or remember some physics, you might remember that velocity is an example of a vector - it has both a speed and a direction.

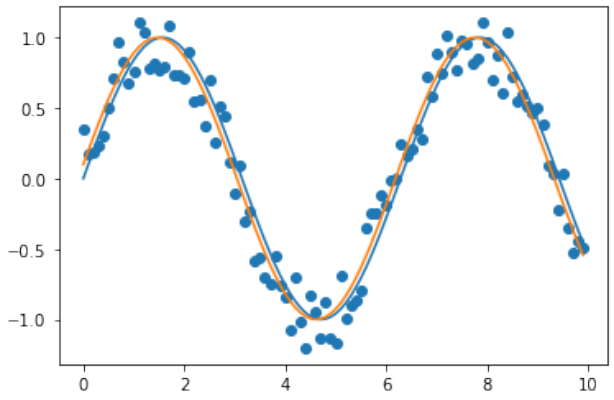



Figure 16.6: Two fits to a set of measured data. It's difficult to tell which is best just by looking, but the sum of squared errors gives us a definitive answer.

In two dimensions this can be represented by an x and y component, and in three dimensions by an x, y and z component. In both cases we can use a one-dimensional array to represent the velocity. [1]

One of the fundamental operations on vectors is the dot product. If you've done a bit of linear algebra, you might have seen it written like this:

$$a \cdot b = ||a||||b|| \cos\theta$$

where $a$ and $b$ are two vectors, $||a||$ is the magnitude of $a$ and $\theta$ is the angle between the two vectors. To compute the dot product, we simply multiply both vectors component-wise and then sum up:

$$a \cdot b = \sum_i a_i b_i$$

where $a_i$ is the $i^{th}$ component of the vector $i$. The dot product can be used to compute the angle between two vectors if we need to do that. If we have two one-dimensional numpy arrays, then we can use `np.dot(a,b)` to compute the dot product.

This gives us an alternative method to computing the sum of squared errors from the previous section. We can think of the $y$ arrays as a series of measurements of y-coordinates, we can think of $y - ymeas$ as a vector in a 100-dimensional space (because there are 100 measurements). We can then simply use `np.dot` on $y - ymeas$ to get the sum-squared error out.

```
x = np.arange(0,10,0.1)
y = np.sin(x+0.1)
error = np.dot(y-ymeas, y-ymeas)
print(error)
```

This is essentially computing the squared magnitude of the difference vector $y - ymeas$. We don't gain very much in this particular case, but we do get a different perspective on what we are computing, and these different perspectives can often be useful.

Vectors can be represented well using one-dimensional arrays. What can two-dimensional arrays be used for? An image is an example of data that can be easily represented using two-dimensional arrays and this is what OpenCV[2] uses to represent images, but we won't explore that here. The fundamental use of two-dimensional arrays is to represent matrices. Matrices form the other core part of linear algebra. Like vectors,

[1] A one-dimensional array can have multiple components so we can easily represent velocity in 2 or 3 dimensions using one-dimensional arrays of shape (2,) and (3,) respectively.

[2] https://opencv.org/. A computer vision and image processing library.

matrices can be added and subtracted together as long as they have the same shape, but the fundamental operation is multiplication. Figure 16.7 gives a general idea of how matrix multiplication occurs (keep in mind that matrices are usually written with indices starting at 1, whereas arrays are written with indices starting at 0). Matrices can only be multiplied if they are of compatible sizes. For $C = AB$, the second dimension of $A$ and the first dimension of $B$ must be equal. In the case of Figure 16.7, $A$ has shape $(4, 2)$ and $B$ has shape $(2, 3)$ and therefore they are compatible. The resultant matrix has a shape consisting of the first and second coordinates of $A$ and $B$ respectively, $(4, 3)$. It is non-trivial, but a good exercise to try and implement the code for matrix multiplication, but luckily the `np.dot` function does all the hard work for us. Creating matrices with actual numbers and performing the multiplication in Python is straightforward:

```
A = np.array(((1,2), (3,4), (5,6), (7,8)))
B = np.array(((9, 10, 11), (12, 13, 14)))
C = np.dot(A, B)
print(A)
print(B)
print(C)
```

which results in:

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
[[ 9 10 11]
 [12 13 14]]
[[ 33  36  39]
 [ 75  82  89]
 [117 128 139]
 [159 174 189]]
```

You should be able to see, that $36 = 1 \times 10 + 2 \times 13$, and $139 = 5 \times 11 + 6 \times 14$. These match up to the computations shown in the figure.

Finally, you should note that vectors can be considered as a matrix with one of its dimensions equal to 1. So we get a column vector when the number of columns is 1; that is, with a shape that looks like $(4, 1)$ or $(350, 1)$. And we get a row vector when the number of rows is 1; that is, with a shape that looks like $(1, 4)$ or $(1, 20)$. In Numpy, a one-dimensional array can substitute as a row or column vector. In either case, we can multiply matrices and vectors using the `np.dot` function. Some examples are given in the exercises.

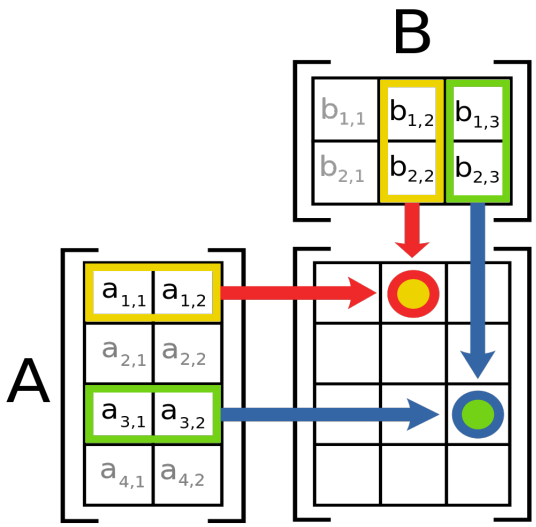



Figure 16.7: An example of multiplying matrices $A$ and $B$. Image downloaded from https://commons.wikimedia.org/wiki/File:Matrix_multiplication_diagram.svg and released under the Creative Commons Attribution-Share Alike 3.0 Unported license https://creativecommons.org/licenses/by-sa/3.0/deed.en

## 16.3 Laboratory Exercises

1. Use `np.arange` to create an array with 100 equally spaced numbers between 0 and 10. Store this array into a variable called `x`.
2. Plot the following functions between 0 and 10:
   (a) $x^2$
   (b) $-3x^3 + 2x^2 - 5$
   (c) $\sin(x)/x$
   (d) $\cos(x)$
   (e) $\log(x)$
   (f) $\exp(x)$
   (g) $\exp\left(\frac{-x^2}{(2\sigma)}\right)$ for various values of $\sigma$ (e.g. $\sigma = 1.0, \sigma = 2.0, \sigma = 10.0$).
3. Write a function that takes two 1D arrays as input and returns the sum squared error between the two arrays. The function should satisfy the following doctests:

```
def sum_squared_error(x1, x2):
    """
    >>> sum_squared_error(np.array([1.0,2.0]), np.array([3.0,4.0]))
    8.0
    >>> sum_squared_error(np.array([1.0,2.0,3.0]), np.array([3.0,4.0,5.0]))
    12.0
    >>> sum_squared_error(np.array([1.0,2.0,3.0,4.0]),
                          np.array([1.0,2.0,3.0,4.0]))
    0.0
    """
```

4. If you didn't use `np.dot` in the previous question redo it using `np.dot`. If you did use `np.dot`, redo it without using it.

5. The classic computer science use of matrix multiplication is using them for rotations of objects either in two or three dimensions. In two dimensions, a rotation matrix looks like this

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

where $\theta$ is amount of rotation to perform given in radians[3]. Note that this is for a rotation around the origin. Given a two-dimensional point, we can compute the rotated point by pre-multiplying the vector representation by the above rotation matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

For example, to rotate[4] the point (1,1) by $\pi/4$ radians (which is $45°$), we perform the following matrix multiplication:

$$\begin{bmatrix} \cos(\pi/4) & -\sin(\pi/4) \\ \sin(\pi/4) & \cos(\pi/4) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1.414 \end{bmatrix}$$

   (a) Write a function that takes as parameters a two-dimensional point (represented by a column vector with two elements, that is a matrix of shape (2,1)), and an angle specified in radians, and returns the point rotated by the given angle. You may want to write a helper function that takes an angle as a parameter and returns the appropriate $2 \times 2$ rotation matrix.

   (b) Write a function called `plot_rotate` that uses the rotate function from the previous question and plots both points using the `plt.scatter` function. Look up the help for `plt.scatter` to find out how to use it (you need to pass an array of $x$-values and an array of $y$-values, so you need to extract the appropriate parts of your point).

   (c) If you've used `np.dot` to create your rotate function, then it should also work for multiple points using exactly the same code as long as multiple points are stored in a $2 \times N$ array (for $N$ points). Try it out by creating an array using this code:

```
pts = np.zeros((2,20))
pts[0,:] = np.arange(-1,1,0.1)
plot_rotate(pts, np.pi/4)
```

   You should get something that looks like Figure 16.8.

[3] There are $2\pi$ radians in $360°$, so to convert from degrees to radians, divide by 360 and multiply by $2\pi$.
[4] Rotations are usually counter-clockwise.

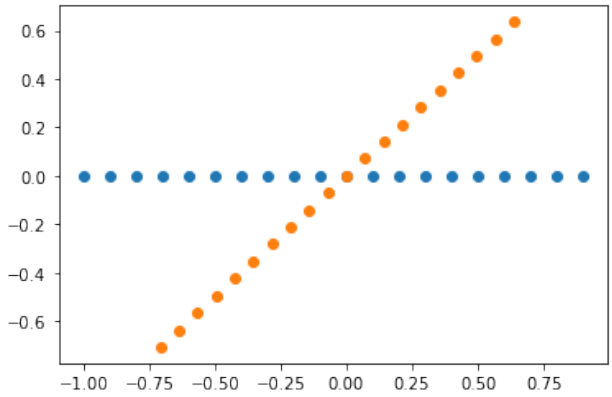



Figure 16.8: Rotating some points using matrix multiplication.

# Lesson 17

# Numpy Case Study

## 17.1 Introduction

In this lesson, we are going to develop an interesting visualisation that makes use of two-dimensional Numpy arrays. This case study is interesting because even though it is quite simple, it exhibits quite complicated behaviour, and, also demonstrates some useful programming practices that Python makes easy to make use of. The example is one of the simplest simulations possible - and can be generalised to be useful in a wide variety of sciences.

## 17.2 Conway's Game of Life

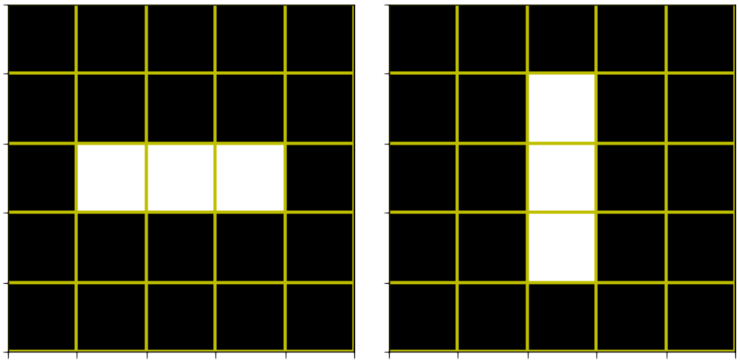
### 17.2.1 The conceptual game

Conway's Game of Life[1] is an example of something called a cellular automata. You can think of cellular automata as a very simple game played alone. The game consists of a board consisting of a grid of cells (hence cellular), a bunch of tokens, and a set of rules for how to update the board. Typically, the tokens are initially placed randomly on the board which is then updated by the rules. In the case of Conway's Game of Life, the rules are applied in time synchronous fashion - that is all cells on the board are updated simultaneously in a single time step. You can also think of cellular automata as the simplest possible simulation that exhibits some of the characteristics of life. Each token represents an organism, each grid cell represents a place in the world that may or may not be occupied, and each time step represents a new generation of organisms with births and deaths or keep living options. The rules for Conway's Game of Life are simple:

1. Any live cell with fewer than two live neighbours dies from starvation.
2. Any live cell with more than three live neighbours dies from overpopulation.
3. Any empty cell with exactly three live neighbours becomes alive by reproduction.
4. Every other cell remains in its current state.

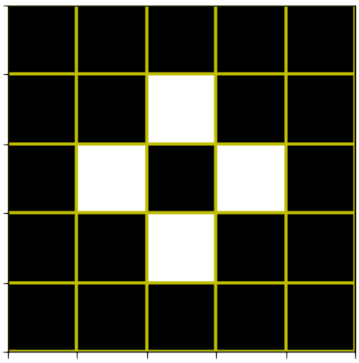
Figure 17.1 shows a simple example on a small $5 \times 5$ grid with a before and after state. If we follow the above rules, we can see that the two outer cells have only one neighbour each and therefore die off in the next generation. The middle cell has two neighbours so stays as is. The cells above and below the middle

[1] Invented by mathematician John Conway in 1969/1970



(a) Generation 1 (b) Generation 2

Figure 17.1: Before and after snapshots of a small Game of Life board after one application of the rules.



(a) Generation 1

Figure 17.2: Generation 1 for a different board configuration. What does Generation 2 look like?

cell (not alive in the before shot), each have three alive neighbours, and therefore become alive in the next generation. What happens in the third generation?

Try finding the next generation for the configuration shown in Figure 17.2.

### 17.2.2 Implementing the game

Let's think about how to implement this game. This problem serves as a great example for program development that is often called top-down design. In this sort of development, we start off with a generic overall view of our program that is written as an algorithm or pseudo-code and progressively refine it until we get to code. At the top level, the problem solution is quite straightforward:

1. For each cell:
2. (a) apply the game of life rules

At each design iteration, we try to be a bit more explicit about the steps involved. For example a next iteration might look like:

1. For each cell:

2. (a) if the cell is alive and has fewer than two neighbours, then kill it
   (b) if the cell is alive and has more than three neighbours, then kill it
   (c) if the cell is dead and has exactly three neighbours, then make it alive
   (d) otherwise leave it the same.

At this point, we cannot proceed further without deciding how we should represent the board. This to-ing and fro-ing between algorithmic decisions and representation (data structure) decisions are absolutely typical of program development. Deciding on an algorithm will often limit the choices of data structure to use, and deciding on a data structure will subsequently limit the implementation of the algorithm. We could use a standard Python data structure to represent the board, such as a list or a dictionary, but neither of these choices is ideal and would be quite inefficient. A two-dimensional Numpy array is an ideal choice because its structure maps directly onto a board (one array element represents one board cell), and it is efficient.

The next algorithm iteration will start looking more like Python code, but not completely so. For example, we can specify the board now, and also how we visit each board cell. Also, let's consider 0 on the board to mean the cell is dead, and 1 to mean the cell is alive. Finally, we'll start off with a random board.

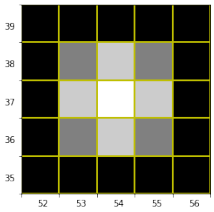
```
numrows = 100
numcols = 100
# create the initial board configuration
board = random.randint(low=0, high=2, size=(numrows,numcols))
# update the board for 100 generations
for generation in range(100):
 # create a board for the next generation - this is quite inefficient (why?)
 newboard = np.zeros(shape=(numrows,numcols))
 # foreach cell in the board
 for row in range(numrows):
  for col in range(numcols):
    # count the number of alive neighbour cells (to be specified)
    num_alive_neighbours = count_num_alive(board, row, col)
    # cell is currently alive
    if board[row,col]==1 and num_alive_neighbours>=2 and num_alive_neighbours<=3:
        newboard[row,col]=1
    # cell is currently dead
    elif board[row,col]==0 and num_alive_neighbours==3:
      newboard[row,col]=1
 board = newboard
 # visualise the board somehow
```

Well, that is pretty much straight Python, but I haven't put everything in. We've got two extra bits to figure out - how to count the number of alive neighbours, and how to visualise the board. This is also typical of top-down design - we can put in place-holders for things we want to figure out a bit later. And it's often the case that we specify those things as functions. Let's start with counting neighbours.

When we're figuring out how to do things, it is often a good idea to go through a specific example on paper first. So let's try and figure out how to access the neighbours of a particular cell. Let's pick the central cell with coordinates $(37, 54)$[2]. What are the coordinates of all it's neighbours? Have a look at Figure 17.3. If we consider only all 8-neighbours, then its neighbours coordinates are:

$$(37, 53), (38, 54), (37, 55), (36, 54), (38, 53), (38, 55), (36, 55), (36, 53).$$

[2]We are indexing the cells by row number first, then column number.




(a) Cell neighbours

Figure 17.3: Consider the white central cell. The light grey cells are its 4-neighbours, and the light grey plus dark grey cells are its 8-neighbours.

What you may notice is that the coordinates are all off by one in either row, column or both, from the original coordinates. In other words, the neighbour coordinates are:

$$(37, 54-1), (37+1, 54), (37, 54+1), (37-1, 54), (37+1, 54-1), (37+1, 54+1), (37-1, 54+1), (37-1, 54-1).$$

Instead of $(37, 54)$, let's consider the generic cell coordinates of interest $(row, col)$. Then the neighbours are:

$$(row, col-1), (row+1, col), (row, col+1), (row-1, col),$$
$$(row+1, col-1), (row+1, col+1), (row-1, col+1), (row-1, col-1).$$

There is one further complication that we need to deal with. What do we consider the neighbours of the cells on the edge of the board? There are two basic choices: either we consider cells outside the board to always be dead (so the coordinate $(-1, 37)$ would implicitly be dead); or we wrap the board around so that the coordinate $(-1, 37)$ is actually $(99, 37)$, and $(100, 37)$ is actually $(0, 37)$ on a $100 \times 100$ board. In the latter case, what we're actually doing is creating a grid on a torus (doughnut) if we wrap around on both the row coordinate and the column coordinate. We can use the modulus operator to give us this wraparound:

$$(row, (col-1)\%numcols), ((row+1)\%numrows, col),$$
$$(row, (col+1)\%numcols), ((row-1)\%numrows, col),$$
$$((row+1)\%numrows, (col-1)\%numcols), ((row+1)\%numrows, (col+1)\%numcols),$$
$$((row-1)\%numrows, (col+1)\%numcols), ((row-1)\%numrows, (col-1)\%numcols).$$

I am going to show you three different ways of implementing the `count_num_alive` function using the basic idea shown above. The first way is what beginner programmers often choose. The method isn't wrong, but it is inelegant, error-prone and inflexible (hard to change). Its advantages are that it is probably the most efficient method of the three, but not by much, and it may be the easiest to understand. It involves an explicit chain of if-statements:

```
def count_num_alive_beginner(board, row, col):
  numrows, numcols = board.shape
  count = 0
  if board[row,(col-1)%numcols]==1:
    count+=1
  if board[row,(col+1)%numcols]==1:
    count+=1
  if board[(row-1)%numrows,col]==1:
    count+=1
  if board[(row+1)%numrows,col]==1:
    count+=1
  if board[(row-1)%numrows,(col-1)%numcols]==1:
    count+=1
  if board[(row+1)%numrows,(col-1)%numcols]==1:
    count+=1
  if board[(row-1)%numrows,(col+1)%numcols]==1:
    count+=1
  if board[(row+1)%numrows,(col+1)%numcols]==1:
    count+=1
  return count
```

But this solution is very ugly and we should always be striving for beauty in our code. It's ugly because it uses a lot more code space than is needed, and it is difficult to modify for different notions of neighbourhood. We can do better. First, note that the offsets rotate through $(-1, 0, 1)$ for both coordinates, so we should be able to replace all those ifs with nested loops like so:

```
def count_num_alive_loops(board, row, col):
  numrows, numcols = board.shape
  count = 0
  for row_offset in range(-1,2):
    nrow = (row+row_offset)%numrows
    for col_offset in range(-1,2):
      ncol = (col+col_offset)%numcols
      count += board[nrow,ncol]
  # need to subtract 1 if the centre pixel is 1
  count -= board[row,col]
  return count
```

That's much more elegant, and I've also managed to eliminate all if-statements (which we could also have done for the previous solution). And notice that the code is quite a bit shorter which is almost always a good thing.

We can't really do any better than that, but we can do it just as elegantly in a slightly different way that allows for more flexibility with defining the neighbourhood. It combines some of the elements of the beginner solution with some of the elements of the loops solution. It involves explicitly creating the offsets and putting them in a list:

```
def count_num_alive_neighbourhood(board, row, col):
  numrows, numcols = board.shape
  count = 0
  neighbours = [(-1,1),(-1,0),(-1,-1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
  for row_offset, col_offset in neighbours:
    nrow = (row+row_offset)%numrows
    ncol = (col+col_offset)%numcols
    count += board[nrow,ncol]
  return count
```

Visualising the board once is straightforward as we can just directly use `plt.imshow`. To visualise more than about 10 is also straightforward, but just using `imshow` will generate a new window for each iteration and the next iteration won't run until the old window is closed. This is useful for debugging, but generally not satisfactory. That code looks like the following:

```
numrows = 100
numcols = 100
# create the initial board configuration
board = random.randint(low=0, high=2, size=(numrows,numcols))
# update the board for 100 generations
for generation in range(10):
 # create a board for the next generation - this is quite inefficient (why?)
 newboard = np.zeros(shape=(numrows,numcols))
 # foreach cell in the board
 for row in range(numrows):
  for col in range(numcols):
    # count the number of alive neighbour cells (to be specified)
    num_alive_neighbours = count_num_alive(board, row, col)
    # cell is currently alive
    if board[row,col]==1 and num_alive_neighbours>=2 and num_alive_neighbours<=3:
        newboard[row,col]=1
    # cell is currently dead
    elif board[row,col]==0 and num_alive_neighbours==3:
      newboard[row,col]=1
 board = newboard
 # visualise the board somehow
 plt.imshow(board)
 plt.show()
```

Matplotlib also has a way of showing animations, but it's not entirely obvious and involves the `FuncAnimation` function. I won't explain how it works here, but here is the code that makes use of the animation and continuously updates the image to show successive generations. I've refactored a bit and created an `game_of_life_update` function that does the heavy lifting of updating the board. The `anim_update` function is the function that handles the animation.

Just a brief overview of how the animation works. First we register the `anim_update` function using the `FuncAnimation` method and we specify the number of frames we want to run (in this case 1000). Matplotlib will then proceed to call `anim_update` 1000 times. Inside `anim_update` we need to access the board and the figure. We do that by specifying that the variable `board` and `ax` are global variables (that is, defined in the main program). Using global variables is a bit of a kludge, and a neater way to do that involves object-oriented programming, but we shan't look at that here. Then we update the board and show the image on the figure and then `FuncAnimation` will call `anim_update` again. Note that this code is a bit inefficient because a new board is created for every iteration - it would be more efficient to have two boards and swap between them.

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as random
import time
import matplotlib.animation as anim

def count_num_alive_neighbourhood(board, row, col):
    numrows, numcols = board.shape
    count = 0
    neighbours = [(-1,1),(-1,0),(-1,-1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
    for row_offset, col_offset in neighbours:
        nrow = (row+row_offset)%numrows
        ncol = (col+col_offset)%numcols
        count += board[nrow,ncol]
    return count

def game_of_life_update(board):
    new_array = np.zeros(board.shape)
    rows = board.shape[0]
    cols = board.shape[1]
    for row in range(rows):
        for col in range(cols):
            num_alive = count_num_alive_neighbourhood(board, row, col)
            if board[row,col]>0 and (num_alive==2 or num_alive==3):
                new_array[row,col]=1
            elif board[row,col]==0 and num_alive==3:
                new_array[row,col]=1

    return new_array

def anim_update(i):
    global board
    global ax
    board = game_of_life_update(board)
    ax.clear()
    ax.imshow(board*255)

board = random.randint(low=0, high=2, size=(100,100))
fig = plt.figure()
ax = fig.add_subplot(111)
a = anim.FuncAnimation(fig, anim_update, frames=1000, repeat=False)
plt.show()
```

## 17.3 Exercises

1. Carefully look at each of the `count_num_alive` variations and write down advantages and disadvantages of each.
2. Type in or download the main game of life code from BlackBoard and run it.
3. Rather than a random board configuration to start, try starting with a single glider/crawler. The initial board configuration for a glider is:

```
board[1,1]=1
board[1,2]=1
board[1,3]=1
board[2,3]=1
board[3,2]=1
```

   With all other cells starting off at zero.

4. If you're feeling adventurous, try some of the patterns from https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.
5. The code outlined in this chapter uses 8-neighbours. Modify the code so it only uses 4-neighbours. The 4-neighbours are usually those that differ by one in one coordinate only. For example, $(0, 1)$ is a 4-neighbour of $(0, 0)$, but $(1, 1)$ is not.
6. Using 4-neighbours and the original rules seems to always very quickly produce a static board. Can you change the rules so the board evolution is more interesting? For example, try changing the number of neighbours needed to procreate or the number of neighbours needed for overpopulation.
7. **Extension Exercise:** As mentioned in the chapter, this code is inefficient because it creates a new board for every iteration. This could get expensive if the board is large (e.g. a $1000 \times 1000$ board takes up 4MB of memory, and 1000 of those is approximately 4GB). Modify the code so that it uses only two boards for the whole length of the program. There are two obvious ways of doing this:
   (a) Copy the contents. Create two board variables called something like `current_board` and `new_board`. At each iteration, set the contents of `new_board` to 0, then update those contents based on the game of life rules using the contents of `current_board` as the base. After updating `new_board`, copy its contents back to `current_board`. This version is still inefficient, because it involves a lot of content copying.
   (b) Swap between two variables. Similar to the previous method except instead of copying contents, we just change what each variable refers to. This is trivial to do in Python. If I have two variables that I want to swap (let's call them $x$ and $y$), then I can just use:

```
x, y = y, x
```

## 17.4 Mastery Level 7 Practice Questions

Mastery level 7 covers Lessons 15 to 17.

1. Plot the following functions between $x = 0$ and $x = 10$ at intervals of $0.1$ (there should be 100 points):
   (a) $y = \sin(x)$
   (b) $y = 3 * \sin(x^2) - 2 * \cos(x^3)$
   (c) $y = 2^{\frac{-x^2}{2}}$

2. More general 2D curves can be represented using a parametric notation with separate equations for the $x$-coordinate and the $y$-coordinate, both of which depend on the same parameter (usually called $t$) which can be thought of as analogous to how far along the curve to go. For example, we can plot a full circle using the following code:

```
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0,2*np.pi,0.01)

x = np.cos(t)
y = np.sin(t)

plt.plot(x,y)
# the following line makes sure the aspect ratio is 1:1
plt.gca().set_aspect('equal')
plt.show()
```

   Plot the following parametric curves:
   (a) A spiral: $x = t\cos(t)$, $y = t\sin(t)$.
   (b) An approximate golden spiral: $x = e^{0.3t}\cos(t)$, $y = e^{0.3t}\sin(t)$.
   (c) A cycloid - trace of a fixed point of a rolling wheel of radius 1: $x = t - \sin(t)$, $y = 1 - \cos(t)$
   (d) A hypocycloid - trace of a fixed point of a rolling wheel of radius b, rolling inside a circle of radius a. $x = (a-b)\cos(t) + b\cos(((a-b)/b) * t)$, $y = (a-b)\sin(t) - b\sin(((a-b)/b) * t)$. Try a bunch of different values of $a$ and $b$ and see what effect it has. Note that any values where $a = 2b$ will result in $y = 0$ for all $t$.

   Figure 17.4 show plots for each of the curves above as well as the circle.

3. 2D rotation matrices were introduced in Exercise 5 on page 205. Another sort of transformation is a scaling transformation which is given by:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

   where $s_x$ and $s_y$ are the scale factors for the $x$ and $y$ coordinates respectively. Apply the scaling matrix with $s_x = 0.5$ and $s_y = 2.0$ to each of the following curves, and plot the results:
   (a) a circle
   (b) a spiral
   (c) a cycloid

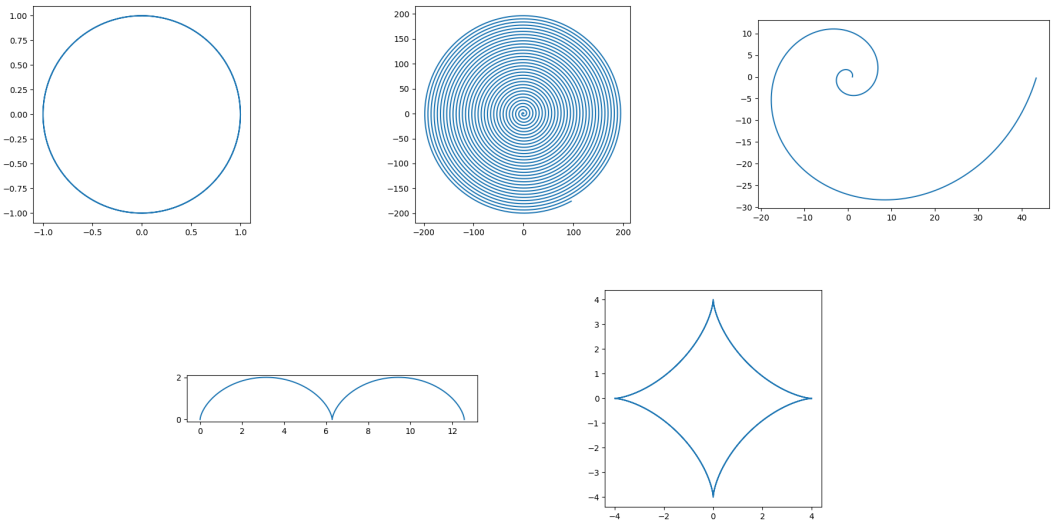


Figure 17.4: A circle, spiral, golden ratio spiral, cycloid and hypocycloid ($a = 4, b = 1$).

Hint: after computing the $x$ and $y$ coordinates, put them into a single matrix with 2 rows and N columns using `np.vstack`:

```
pts = np.vstack((x,y))
```

and then premultiply the points array by the scaling matrix. Your plots should look similar to Figure 17.5.

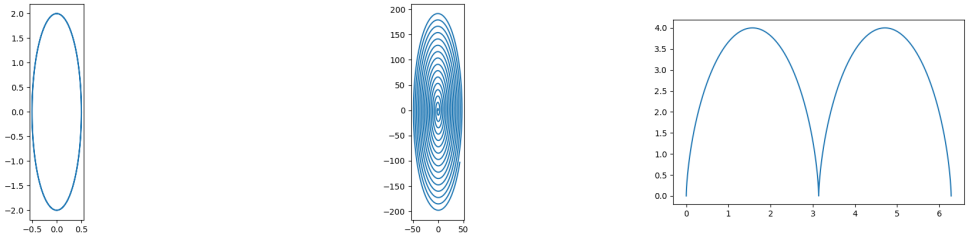


Figure 17.5: A scaled circle, spiral and cycloid.

4. More complicated transformations can be created by applying several different transformations to our pointset of interest. For example, if we have a set of points in the 2D matrix P (2 rows, N columns), and we want to first scale the points and then rotate them, then we can do the following:

$$P' = RSP$$

where $P'$ are our new points, and the right-hand side involves multiplying the matrices R, S and P. Since matrix multiplication is associative, we either do $(RS)P$ or $R(SP)$. But the former is more efficient because it is only accessing all the points once (and $N$ can be very large). Write programs for first scaling the points by $s_x = 1, s_y = 3$, and then rotating by $\pi/4$ radians for each of the following curves:

(a) a circle

(b) a spiral

(c) a cycloid.

Your plots should look similar to those shown in Figure 17.6.

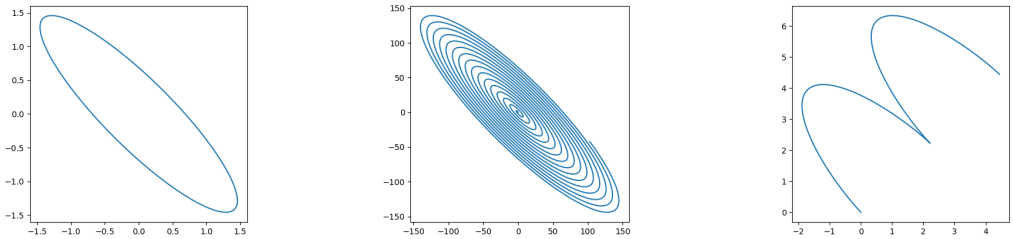



Figure 17.6: A scaled and rotated circle, spiral and cycloid.

# Lesson 18

# Pandas

## 18.1 Introduction

Numpy is great for dealing with numeric data that is all the same type (e.g. float) but is less good at handling heterogeneous data, and is not good at all in dealing with meta-data (information about the data). Other languages and systems, such as R, SAS, and even Excel, are very good at dealing with heterogeneous data and meta-data. Pandas is a library for Python that takes the best ideas from those other systems, incorporates them into the Python environment, and improves on them. The name Pandas is a shortening of "PANel DAta", which is a common term from econometrics for the sort of heterogeneous data we see in spreadsheets. Given that Pandas contains much of the functionality of other data analysis systems, and Python is a more flexible and easy to use language and environment, there doesn't seem to be a good reason for using any other platforms, except for one. Python is easier to use in general than R, but R has a larger statistical community and has more support for a very wide range of statistical methods through its user contributions (CRAN). However, for common statistical and data analysis tasks, Pandas is a very good choice.

Before doing anything else, you should install Pandas as a library using the Thonny package manager (Tools → Manage Packages). Note that when importing Pandas it is usual to import it as:

```
import pandas as pd
```

and then use `pd` throughout your code.

## 18.2 The Data Frame

In Pandas (and in R), the main data structure is the data frame. This is similar to a two-dimensional numpy array, but with better handling of heterogeneous data and meta-data. Figure 18.1 gives a simple graphical overview of the data frame. The easiest way to create a data frame is to read in a data file. Pandas has facilities for reading in several data file types including csv (`read_csv`), excel (`read_xls`), json[1] (`read_json`), html (`read_html`) and others. We are going to stick with using csv files. Download the greenhouse gas emissions csv from the COMP151 blackboard page and save the file into the same directory as your source code. This file contains the data on greenhouse gases for NZ up to 2018, and will provide an interesting use case for investigating the capabilities of pandas.

[1] javascript object notation

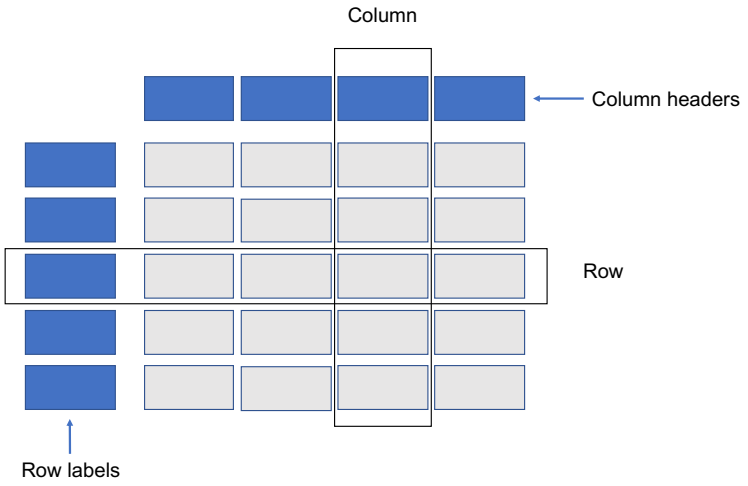



Figure 18.1: A pictorial representation of a data frame. The column headers and row labels are an integral part of the data structure and makes referring to parts of the data frame straightforward.

```
import pandas as pd

ghg = pd.read_csv('greenhouse-gas-emissions.csv')
print(ghg)
```

which should produce something like:

```
          region anzsic_descriptor                          gas       units  \
0       Auckland       Agriculture  Carbon dioxide equivalents  Kilotonnes
1       Auckland       Agriculture  Carbon dioxide equivalents  Kilotonnes
2       Auckland       Agriculture  Carbon dioxide equivalents  Kilotonnes
3       Auckland       Agriculture  Carbon dioxide equivalents  Kilotonnes
4       Auckland       Agriculture  Carbon dioxide equivalents  Kilotonnes
...
                       magnitude  year  data_val
0     Carbon dioxide equivalents  2007    811.63
1     Carbon dioxide equivalents  2008    777.95
2     Carbon dioxide equivalents  2009    684.00
3     Carbon dioxide equivalents  2010    706.56
4     Carbon dioxide equivalents  2011    764.30
```

The output is actually wrapped over multiple lines so the first few columns are at the top and last few columns are at the bottom.

Whereas reading files is achieved by the `read_*` functions, writing files can be done by the equivalent `to_*` functions. For example, if I wanted to save the greenhouse gas emissions data to an excel file, I can use the following code[2]:

```
ghg.to_excel('greenhouse_gases.xlsx', sheet_name="By region")
```

Which is very useful if you need to convert one data format to another - you can also use Excel to convert from csv to xlsx and vice versa, but Excel has fewer options around other file types.

[2]first you will need to install the `openpyxl` package using "Tools → Manage Packages "

## 18.3 Extracting Columns

Extracting columns from a data frame is very easy – we can use the column name inside square brackets. For example, to extract the region column, we can use:

```
print(ghg['region'])
```

which produces:

```
0         Auckland
1         Auckland
2         Auckland
3         Auckland
4         Auckland
           ...
3055    West Coast
3056    West Coast
3057    West Coast
3058    West Coast
3059    West Coast
Name: region, Length: 3060, dtype: object
```

This has many repeats of the same region which may not be what we want. To get just a single copy of each particular region, we can use the `unique` method:

```
print(ghg['region'].unique())
```

which returns a list of unique region names:

```
['Auckland' 'Bay of Plenty' 'Canterbury' 'Gisborne' "Hawke's Bay"
 'Manawatu-Whanganui' 'Marlborough' 'Northland' 'Otago' 'Southland'
 'Taranaki' 'Tasman/Nelson' 'Waikato' 'Wellington' 'West Coast']
```

To extract more than one column, we can use a comma separated list of column names:

```
print(ghg[['region','data_val']])
```

Note the double set of square brackets – the input to the `ghg` square bracket operator is actually a list of column names, and the inner set of square brackets denotes the list. In any case, the result is:

```
          region  data_val
0       Auckland    811.63
1       Auckland    777.95
2       Auckland    684.00
3       Auckland    706.56
4       Auckland    764.30
...          ...       ...
3055  West Coast   1236.64
3056  West Coast    587.02
3057  West Coast    494.47
3058  West Coast    135.64
3059  West Coast     19.52

[3060 rows x 2 columns]
```

Each column of a data frame is called a series. A series is a one-dimensional array and has a single data type (is homogeneous).

## 18.4 Extracting Rows

There are three common ways of extracting rows from a data frame in Pandas: label-based row indexing (loc), position-based row indexing (iloc), and row filtering. I cover each of these in the following. There are other methods, but these are the most common.

### 18.4.1 Label-based row indexing

Associated with each row of a data frame is a row label. Quite often, and by default, this label is an integer, but it can be also be a string label[3]. It is often unique, but it need not be. The greenhouse gas data uses integer labels, so to make things a bit clearer, let's create a simpler data frame with strings as row labels. We can do this directly using the following technique:

```
    d = {'col1': [1, 2, 3], 'col2': [3, 4, 5]}
    df = pd.DataFrame(d, index=['rowa', 'rowb', 'rowc'])
    print(df)
```

which produces:

```
      col1  col2
rowa     1     3
rowb     2     4
rowc     3     5
```

The `d` variable in the above code is an example of a Python dictionary, and is one way we can create data frames from scratch in our code. A dictionary is a very useful data structure that allows for key-based indexing where a key is often a number or a string. Data frames act a bit like dictionaries - `df['col1']` returns a Pandas series, whereas `d['col1']` returns a list, but the series and list have the same contents (`[1, 2]`). In any case, to access a specific row, we can use the `loc` attribute:

```
>>> df.loc['rowb']
col1    2
col2    4
Name: rowb, dtype: int64
```

If we extract one row, the result is a Series. If we extract more than one row, the result is a data frame:

```
>>> print(df.loc['rowb':'rowc'])
      col1  col2
rowb     2     4
rowc     3     5
```

### 18.4.2 Position-based row indexing

Position-based row indexing is very similar to row-based indexing, except that we must use the (0-based) row number to indicate a row, and we use the `iloc` attribute:

[3] in fact, any object, but integers and strings are the most common

```
>>> df.iloc[1]
col1    2
col2    4
Name: rowb, dtype: int64
```

We can also choose multiple rows using slicing:

```
>>> print(df.iloc[1:3])
      col1  col2
rowb     2     4
rowc     3     5
```

### 18.4.3 Row filtering

Row filtering is a way of extracting rows from the data frame based on their contents. In some ways it is the most interesting and powerful way of selecting parts of a data frame. For example, if we want all the rows from Otago, we would do the following:

```
>>> print(ghg[ghg['region]=='Otago'])
     region anzsic_descriptor                          gas       units  \
96    Otago       Agriculture  Carbon dioxide equivalents  Kilotonnes
97    Otago       Agriculture  Carbon dioxide equivalents  Kilotonnes
98    Otago       Agriculture  Carbon dioxide equivalents  Kilotonnes
99    Otago       Agriculture  Carbon dioxide equivalents  Kilotonnes
100   Otago       Agriculture  Carbon dioxide equivalents  Kilotonnes
...     ...               ...                         ...         ...
```

There's a technicality here that is worth mentioning. The term `ghg['region']=='Otago'` is actually a valid expression whose value is a Pandas Series of booleans:

```
>>> ghg['region']=='Otago'
0       False
1       False
2       False
3       False
4       False
        ...
3055    False
3056    False
3057    False
3058    False
3059    False
Name: region, Length: 3060, dtype: bool
```

The same sort of indexing can be done with numpy and numpy arrays, although we have not discussed it in that context. The result of indexing a Series with a boolean array is to select out the rows of the data frame for which the value in the array is True. As a consequence, we can use any valid boolean expression as the index as long as the length of the resulting array is the same as the number of rows in the data frame. Well, almost. There is a special format for more complicated boolean expressions: each index expression must be surrounded by brackets, and we need to use the special boolean operators & for `and` and | for `or`. For example, if we want to find those industries in Otago that produce a large amount of greenhouse gases:

```
>>> otago_ghg = ghg[(ghg['region']=='Otago') & (ghg['data_val']>2000)]
>>> print(otago_ghg[['anzsic_descriptor', 'year', 'data_val']])
       anzsic_descriptor  year  data_val
96           Agriculture  2007   3547.43
97           Agriculture  2008   3388.17
98           Agriculture  2009   3423.31
99           Agriculture  2010   3432.29
100          Agriculture  2011   3574.17
101          Agriculture  2012   3752.09
102          Agriculture  2013   3784.08
103          Agriculture  2014   3903.89
104          Agriculture  2015   3860.06
105          Agriculture  2016   3507.22
106          Agriculture  2017   3512.53
107          Agriculture  2018   3733.49
1536  Primary industries  2007   3642.48
1537  Primary industries  2008   3493.38
1538  Primary industries  2009   3568.60
1539  Primary industries  2010   3576.39
1540  Primary industries  2011   3700.80
1541  Primary industries  2012   3883.93
1542  Primary industries  2013   3932.26
1543  Primary industries  2014   4016.73
1544  Primary industries  2015   3970.40
1545  Primary industries  2016   3625.10
1546  Primary industries  2017   3638.14
1547  Primary industries  2018   3860.07
...
```

Here I've left out the rows at the end that are various totals. Note that the primary industries output includes agriculture, so in Otago, most primary industries are agriculture.

## 18.5 Exercises

1. Download the greenhouse gas emissions csv file. Write a script (or do it in the shell) to read in the file using `read_csv`, and print it out.
2. Try saving the data frame to an excel file and then open that file using excel.
3. Print out all the unique `anzsic_descriptor`s.
4. Use the web to find out what an `anzsic_descriptor` is.
5. Print out the greenhouse gas emissions associated with households in NZ.
6. Use the `sum` function to find the total greenhouse gases produced by households in NZ for 2007 to 2018.
7. Use the `sum` function to find the total greenhouse gases produced by households in NZ for 2007 only.
8. Repeat the previous question for each year up until 2018. What can you conclude?
9. Repeat the previous question for each region. *Hint:* Now is a good time to make use of the skills you've acquired from previous chapters and in particular the for loop. `ghg['region'].unique()` gives us a list of regions and we can loop over that list using:

```
for region in ghg['region'].unique():
    # do something with the variable region
```

   Your job here is to fill in the `# do something with ...` bits of the code.

# Lesson 19

# Pandas Part 2

## 19.1 Summary Statistics

Pandas is a library for handling data and doing data analysis and statistics. It should be no surprise that there are quite a few statistical functions provided with Pandas. Here is a list of the common summary statistics that people often use:

| Function | Description |
|---|---|
| count | Number of non-NA observations |
| sum | Sum of values |
| mean | Mean of values |
| mad | Mean absolute deviation |
| median | Arithmetic median of values |
| min | Minimum |
| max | Maximum |
| mode | Mode |
| abs | Absolute Value |
| prod | Product of values |
| std | Bessel-corrected sample standard deviation |
| var | Unbiased variance |
| sem | Standard error of the mean |
| skew | Sample skewness (3rd moment) |
| kurt | Sample kurtosis (4th moment) |
| quantile | Sample quantile (value at %) |
| cumsum | Cumulative sum |
| cumprod | Cumulative product |
| cummax | Cumulative maximum |
| cummin | Cumulative minimum |

These functions (methods really) can be applied either to data frames or to series. In a data frame, Pandas automatically only applies the method to the columns for which it makes sense to do so. But there are sum[1] surprises. For example, if we take the greenhouse gas data from the previous chapter, loaded into the variable `ghg`, and apply the count method:

[1] pun intended

```
>>> ghg.count()
region                3060
anzsic_descriptor     3060
gas                   3060
units                 3060
magnitude             3060
year                  3060
data_val              3060
dtype: int64
```

which makes perfect sense. There are 3060 rows in the data. If we compute the mean, then we get:

```
>>> ghg.mean()
year        2012.500000
data_val    1464.322915
dtype: float64
```

The `mean` method is only applied to the numeric columns. Of course, it doesn't really make sense to find the mean of the year column, so it's probably best just to apply it to the particular column of interest:

```
>>> ghg['data_val'].mean()
1464.3229150326795
```

Here's the sum surprise:

```
>>> ghg.sum()
region                AucklandAucklandAucklandAucklandAucklandAuckla...
anzsic_descriptor     AgricultureAgricultureAgricultureAgricultureAg...
gas                   Carbon dioxide equivalentsCarbon dioxide equiv...
units                 KilotonnesKilotonnesKilotonnesKilotonnesKiloto...
magnitude             Carbon dioxide equivalentsCarbon dioxide equiv...
year                                                        6158250
data_val                                                 4480828.12
dtype: object
```

The first five columns contain strings to describe various things. But since we can add two strings together (python concatenates), we can also apply sum[2] to a bunch of strings, and the result is the concatenation of all of them. This is almost certainly not what we want in this case, so again, it's usually better to apply sum only to the columns we want.

These methods are all very useful, but Pandas has an extremely useful convenience function that computes a few of these for us and puts the results in a new data frame. It's called `describe`:

```
>>> ghg.describe()
             year           data_val
count     3060.000000    3060.000000
mean      2012.500000    1464.322915
std          3.452617       2353.150888
min       2007.000000    4.550000
25%       2009.750000    101.045000
50%       2012.500000    409.695000
75%       2015.250000    1629.117500
max       2018.000000    15660.240000
```

[2]Note that this doesn't work for Python's inbuilt sum function and strings, only for the Pandas sum method.

## 19.2 Plotting Data

For any sort of data analysis, the first thing you should always do is look at the data, and make sure it makes sense. It's very easy to forget this step, but that is almost always a mistake. Depending on the size of the data, one can look at the actual numbers, but it's often better to look at a visual representation of the data. Underlying the Pandas plotting functionality is a python library called `matplotlib`, which can be a bit more flexible for plotting than Pandas, but comes with associated extra complexity. We will just focus on Pandas plotting in COMP151. There are many different types of plots and visualisations that can be done, and we'll cover some fundamental rules of visualisation in a later lecture. For now, we are going to look at some basic types of plots: line plots, bar plots, and box plots.

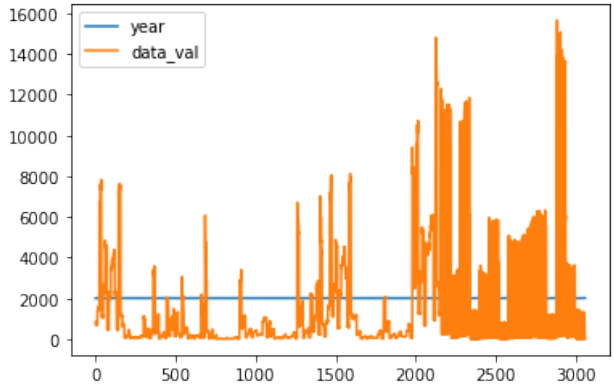
### 19.2.1 Line Plots

Line plots are the fundamental plot in Pandas. In a script, or in the Thonny shell, we also have to use the `matplotlib show` function as is shown in the following code, for the plot to appear in a new window. From now on, I'm just going to assume that the `plt.show` function is called after each plot call.

```
import matplotlib.pyplot as plt

ghg.plot()
plt.show()
```

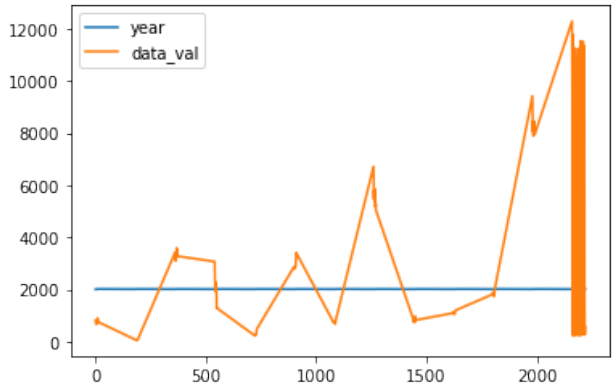
The default line plot plots each numeric column in a data frame with its own line:




But this is not a sensible way of plotting the data, because each row is a mix of different regions and some rows are totals. It also makes no sense to plot the year column. So let's select out some data first. For example, how does Auckland go for greenhouse gas emissions over time?

```
>>> ghg[ghg['region']=='Auckland].plot()
```

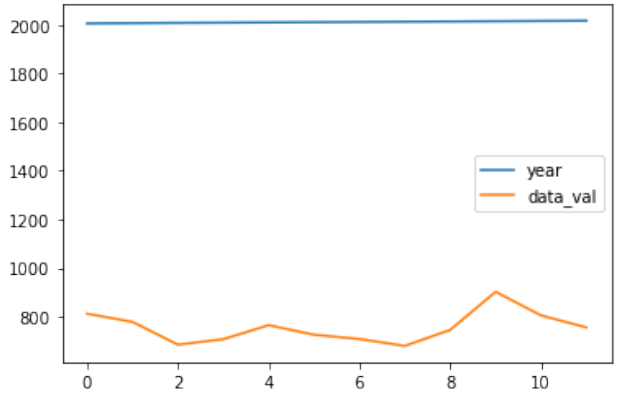
produces:




Which is better, but still includes weird total rows. We can eliminate those, by choosing the rows with descriptor 'Agriculture':

```
>>> ghg[(ghg['region']=='Auckland') & (ghg['anzsic_descriptor']=='Agriculture')].plot()
```

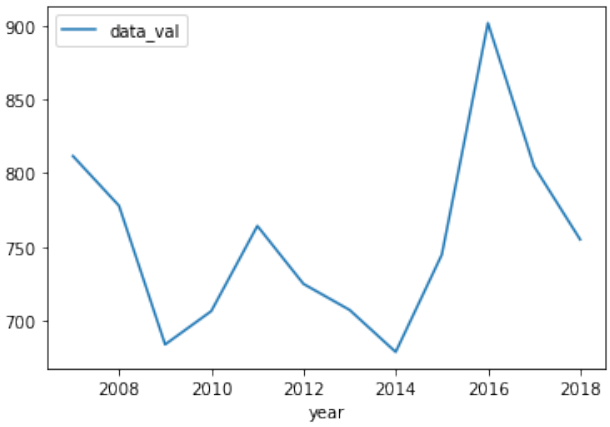
produces:




And now we're getting somewhere. Still, we really don't want the year plotted as a separate line. What we really want is the year on the x-axis and the data value on the y-axis, luckily, that's pretty easy to do:

```
>>> ghg[(ghg['region']=='Auckland') &
        (ghg['anzsic_descriptor']=='Agriculture')].plot(x='year',y='data_val')
```

produces:




Which is pretty much what we're looking for.

Finally, let's say we want to plot all the year data from each region on the same graph so we can get a bit of an idea about each region and also compare regions. In this case, we have a much more difficult problem because the data isn't in a format that Pandas can easily plot. However, these sorts of problems are very common. Quite often we need to transform the format of the data into a more amenable form, for future processing. This is where the power of programming really comes into its own. Up until now, the problems we have dealt with using Pandas have only required rudimentary programming skills, but quite good knowledge of the Pandas library. As we've seen above, Pandas naturally plots multiple columns on the same graph (see the plots that plotted the data value and the year). But our base data has all the important data in the same column, and there is a sequence of rows that represent each region. If I want to plot all regions on the same graph, one way is to create a new data frame that has a single year as a row, and each region as a column. In the code from the previous paragraph, we extracted out the rows relevant to Auckland. What we need to do, is repeat that process for each of the regions. We could decide to do this manually, by directly editing the data file, or by typing in the above code for each of the regions, and both of those options can be a good idea at times when the number of data are small, but even then such a manual process is error prone. If we had a large data set, with very many regions, then a manual process will take too long, be very boring, and almost certainly introduce errors. That's where programming excels (at long, boring and error-prone tasks).

It's worth thinking about an algorithm (an English-like description of a solution) for achieving our goal. In this case, we need to:

1. create a new data frame with all the requisite columns (Auckland, Otago, etc), ideally using the year as the row index

2. for each region, extract the appropriate data from the original data frame and assign it to the appropriate column of the new data frame.

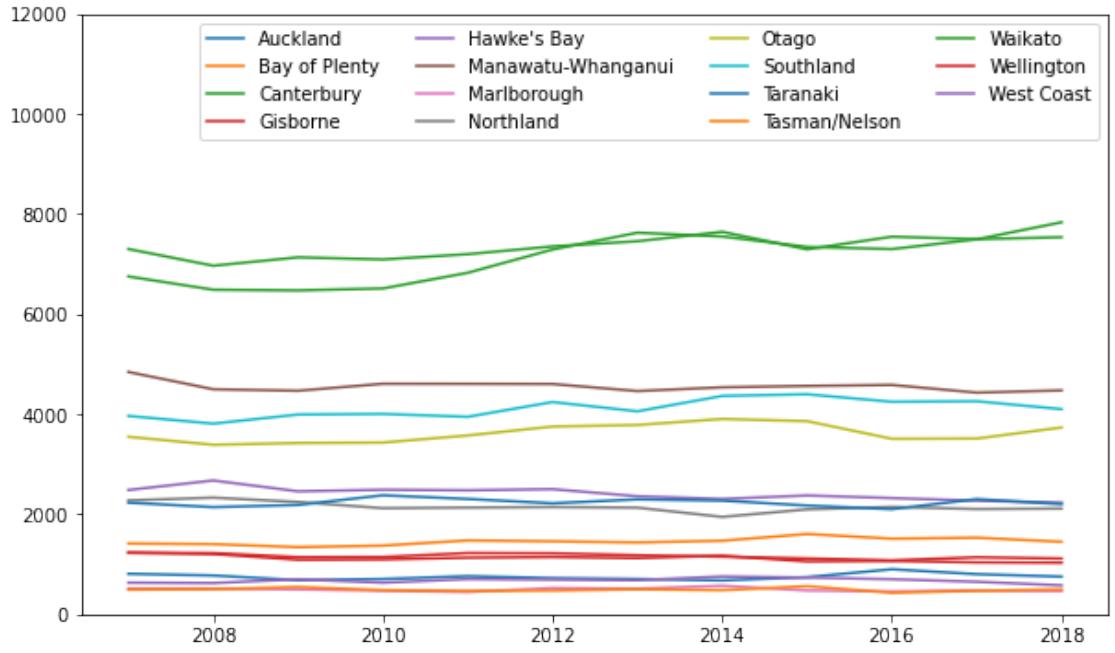
The Python code looks very similar to the above algorithm in structure:

```
# we need matplotlib to move the legend
import matplotlib.pyplot as plt

# first find the names of all the regions
regions = ghg['region'].unique()
# also find each of the years
years = ghg['year'].unique()
# create a new DataFrame with the rows indexed by the years,
# and one column for each region
regions_df = pd.DataFrame(index=years, columns=regions)
# now fill in the data for each column in the DataFrame
for region in regions:
    # this is very similar to the code above for Auckland
    # except we use the region variable
    # instead of the 'Auckland' literal
    regions_df[region] = \
        ghg[(ghg['region']==region) &
            (ghg['anzsic_descriptor']=='Agriculture')]['data_val'].values

# make the plot bigger, and set the y axis limits
regions_df.plot(figsize=(10,6), ylim=(0,12000))
# this fixes problems with the legend
plt.legend(ncol=4)
```

And the output is a pretty graph:



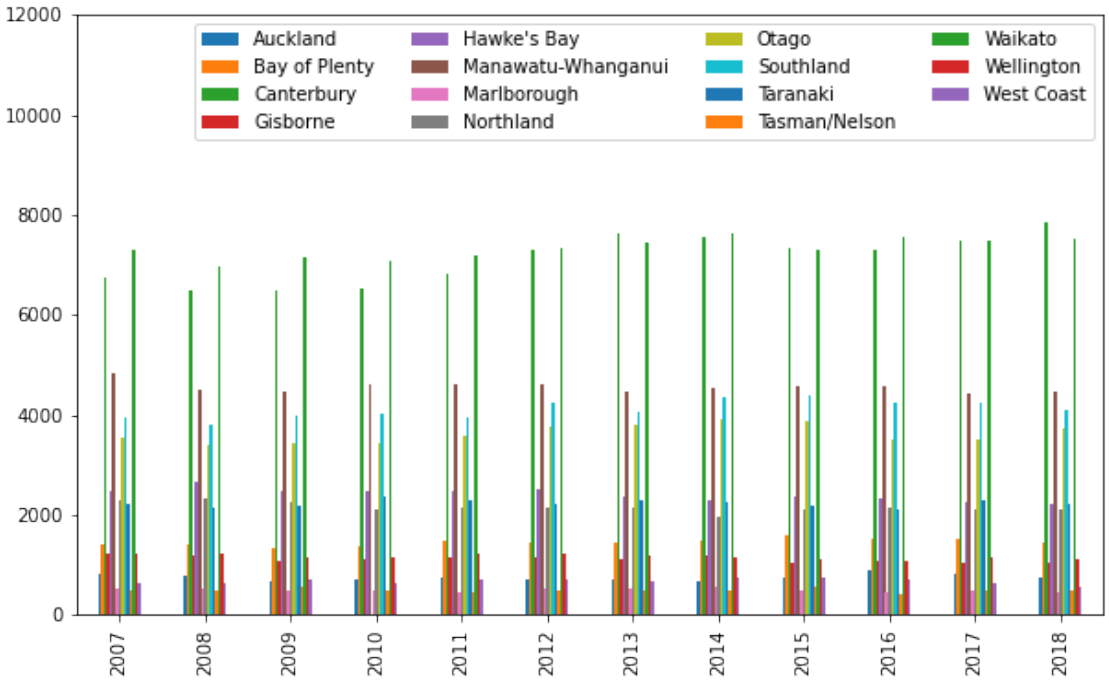

### 19.2.2 Bar Graphs

We can see from the previous graph that greenhouse gas emissions have not reduced for any region in NZ, or at least not by much, which is a bit depressing. Also note that graphs like this that rely on colour to

convey information are very difficult for colour-blind people to interpret. A different way of plotting this data is using a bar graph:

```
regions_df.plot(kind='bar', ylim=(0,12000), figsize=(10,6))
plt.legend(ncol=4)
```

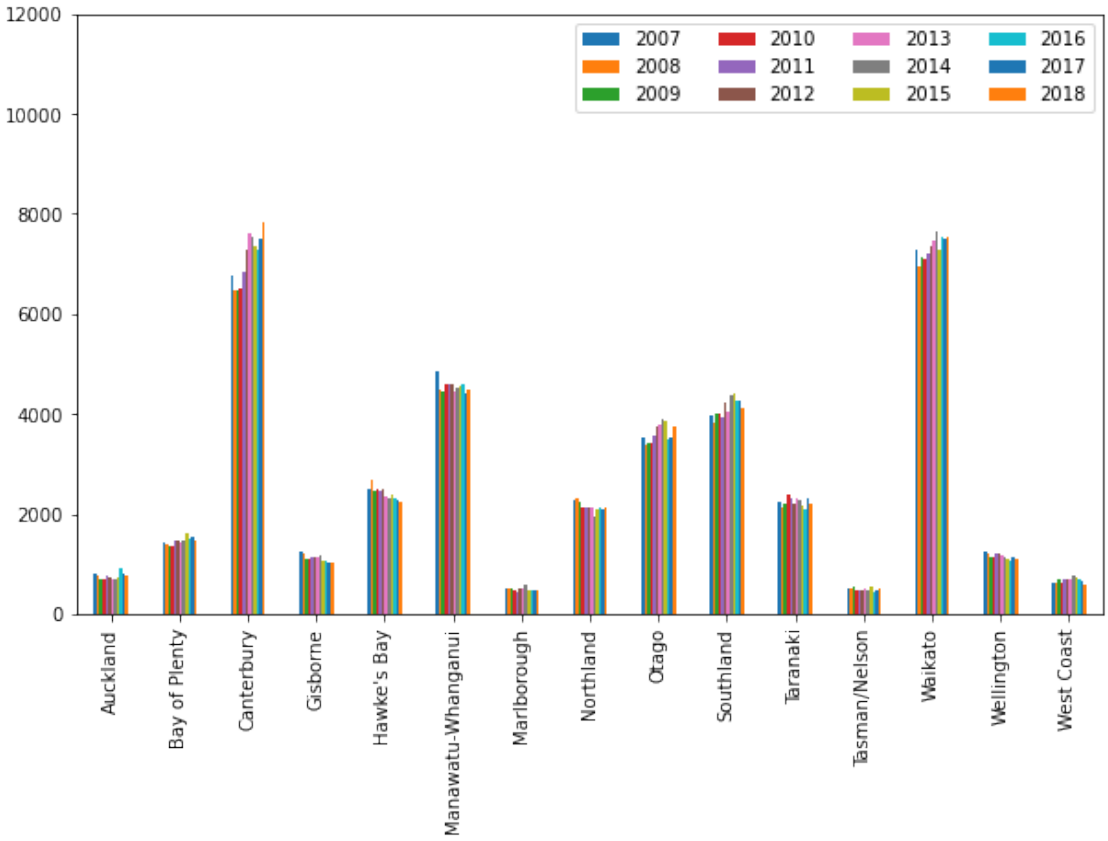
But the result is not very illuminating or clearer than the line plot:




We really want the plot around the other way, with the region being along the rows and the year along the columns of the data frame. We could redo the code above, but there is an easier way that swaps rows and columns, and it is called a transpose:

```
regions_df.T.plot(kind='bar', ylim=(0,12000), figsize=(10,6))
plt.legend(ncol=4)
```

The attribute `regions_df.T` returns a new data frame that is the transpose of the original, and this produces the following graph:



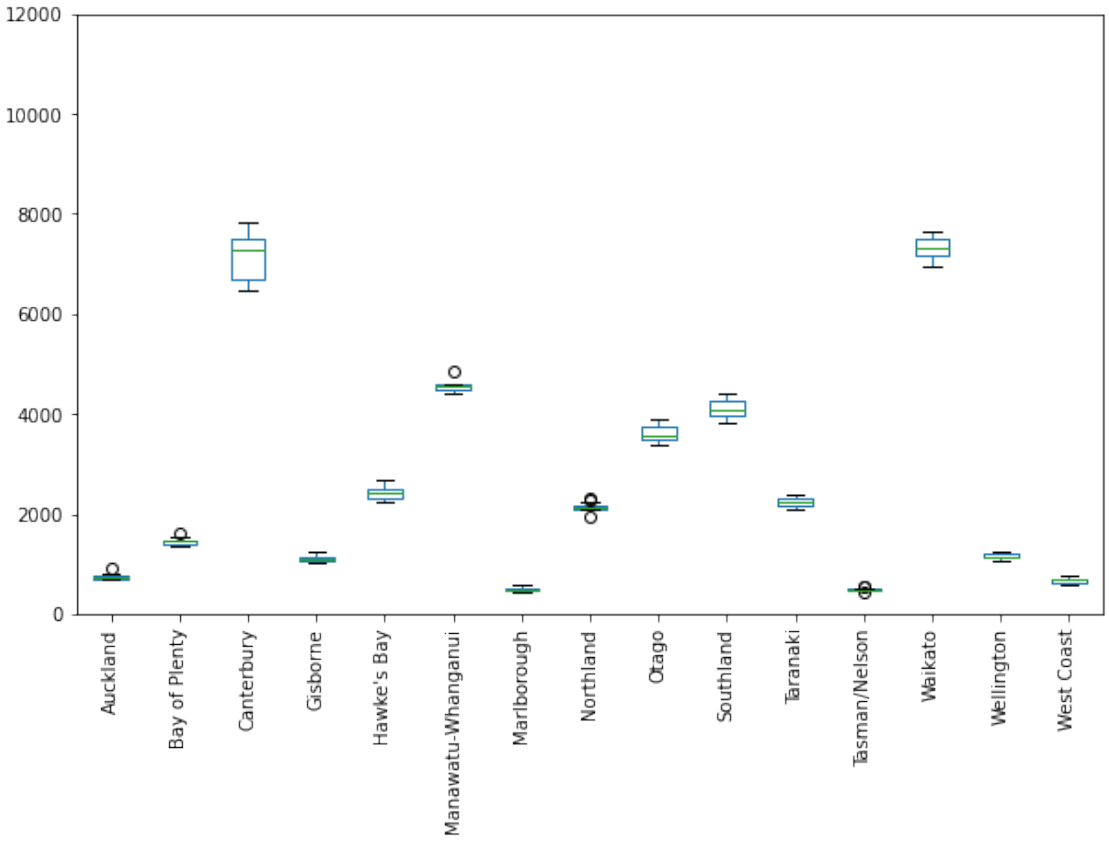

It is now clear that Canterbury and Waikato are the biggest producers, presumably a combination of physical size, population and intensive farming practices. It is also clear that no region is effectively reducing greenhouse gas emissions. But compare how easy it is to interpret the second bar graph, compared to the other two types of plots.

### 19.2.3 Box Plots

The final plot we are going to touch on today is the box plot. The bar plot in the previous graph gave us a pretty good indication of the change over the years for each region. We can compress that representation to give an idea of how much variation there was from year to year at the expense of losing the information about how the data changes over time. A box plot indicates the median, the 25th and 75th percentiles, the overall range of the data, and any outliers (points deemed to be outside the usual range of points):

```
regions_df.plot(kind='box', ylim=(0,12000), figsize=(10,6))
# the following command rotates the x labels so we can see them
plt.xticks(rotation=90)
```

producing the following graph:




Although not necessarily relevant for this data, box plots can give the viewer a reasonable idea if one set of data really is different to another, based on whether the boxes overlap on the graph. For example, from the graph we can deduce that Canterbury and Waikato probably do not produce a different amount of greenhouse gases, but it looks like Southland really does produce a bit more than Otago. We will see a bit more of how to answer such questions when we look at statistical tests in a future lecture.

## 19.3 Exercises

To complete these exercises, you should load in the renewable energy stocks csv file from the COMP151 Blackboard page.

1. Report (print out) the total number of GigaWatt hours generated for each type of resource over the whole data set. That is, what's the total for Biogas, Gas, Coal, Wind etc.
2. Report the mean number of Gigawatt hours generated for each year.
3. Report the mean number of Gigawatt hours generated for each resource. That is, the mean for Biogas, Gas, etc.
4. Produce a line plot for the number of Gigawatt hours generated each year for Hydro.
5. Produce a line plot for the number of Gigawatt hours generated each year for each of the types of resource.
6. Produce a bar graph showing the progression of each resource by year. That is, one resource should be in a bar group, and each year should be plotted within that group, and there should be a group for each resource.
7. Repeat the previous question, but this time produce a boxplot for each resource.

## 19.4 Mastery Level 8 Practice Questions

Mastery level 8 covers Lessons 18 and 19.

1. Write a function that reads in the Tara Hills Temperature data and prints out the mean temperature for each month for a specified range of years (including the start year, and up to but not including the end year). The filename, start and end year should be passed in as parameters to the function. Your function should pass the following doctests.

```
import pandas as pd

def mean_temp(filename, start_year, end_year):
    """
    >>> mean_temp('Tara_Hills_temp.csv', 1950, 1960)
    JAN mean = 15.834
    FEB mean = 15.488
    MAR mean = 13.317999999999998
    APR mean = 9.615
    MAY mean = 5.385
    JUN mean = 2.636
    JUL mean = 1.441
    AUG mean = 3.7211111111111115
    SEP mean = 7.514999999999999
    OCT mean = 9.839
    NOV mean = 12.228
    DEC mean = 14.162
    >>> mean_temp('Tara_Hills_temp.csv', 1990, 1999)
    JAN mean = 16.158571428571427
    FEB mean = 16.067777777777778
    MAR mean = 12.4075
    APR mean = 9.087142857142858
    MAY mean = 6.363333333333333
    JUN mean = 2.9425
    JUL mean = 2.565
    AUG mean = 4.207777777777778
    SEP mean = 6.883333333333334
    OCT mean = 9.867142857142856
    NOV mean = 11.46375
    DEC mean = 14.22625
    """

if __name__=='__main__':
    import doctest
    doctest.testmod(verbose=True)
```

2. Write a function that reads in the Tara Hills Temperature data and returns the mean temperature for a given year for a specified range of months (including the first month, and up to and including the end month). The filename, start and end month should be passed in as parameters to the function. Your function should pass the following doctests.

```
import pandas as pd

def mean_temp(filename, year, start_month, end_month):
    """
    >>> mean_temp('Tara_Hills_temp.csv', 1950, 'FEB', 'JUN')
    8.97
    >>> mean_temp('Tara_Hills_temp.csv', 1973, 'SEP', 'DEC')
    11.5175
    """

if __name__=='__main__':
    import doctest
    doctest.testmod(verbose=True)
```

3. Write a program that reads in the effects of covid 19 on trade data file and prints out the average value of each type of trade (Direction column) for each year in the data set. The output of your program should look like:

```
Exports 2015 19791628.138974886
Exports 2016 19505767.308022678
Exports 2017 21688296.64371773
Exports 2018 23139822.86745125
Exports 2019 24181366.440415844
Exports 2020 24208945.627898987
Exports 2021 23168046.34513941
Imports 2015 44308094.64508095
Imports 2016 43853343.276162066
Imports 2017 48047120.418848164
Imports 2018 53207274.539113104
Imports 2019 54368368.61768369
Imports 2020 48794088.425235964
Imports 2021 52639163.91639164
Reimports 2015 640883.9779005524
Reimports 2016 539944.9035812672
Reimports 2017 469780.2197802198
Reimports 2018 486263.73626373627
Reimports 2019 454545.45454545453
Reimports 2020 545454.5454545454
Reimports 2021 518518.51851851854
```

4. Write a program that reads in the effects of covid 19 on trade data file and prints out the average value of each year for each weekday in the data set. The output of your program should look like:

```
2015 Thursday 29465064.030941125
2015 Friday 29289744.20638391
2015 Saturday 17426155.809859157
2015 Sunday 16651034.558180228
2015 Monday 34703606.47135986
2015 Tuesday 26867431.69877408
2015 Wednesday 24016954.022988506
```

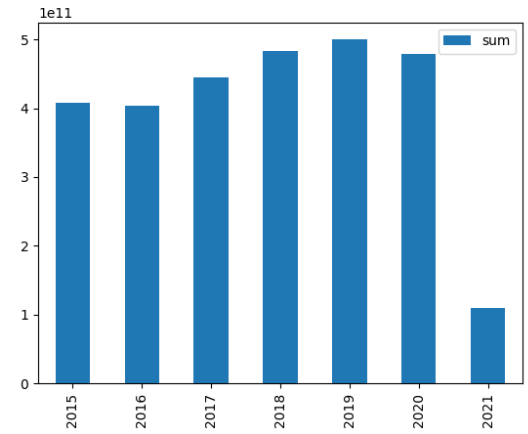
```
2016 Thursday 30912017.974572554
2016 Friday 29913135.19313305
2016 Saturday 16620112.121212121
2016 Sunday 18480517.95096322
2016 Monday 30116330.114135206
2016 Tuesday 26130385.35729943
2016 Wednesday 24154529.151943464
2017 Thursday 31537727.192982458
2017 Friday 32130903.846153848
2017 Saturday 16655433.274802458
2017 Sunday 21418299.69944182
2017 Monday 36833901.3215859
2017 Tuesday 26404778.07017544
2017 Wednesday 29887371.8061674
2018 Thursday 35091547.72329247
2018 Friday 34534887.237762235
2018 Saturday 15008951.499118166
2018 Sunday 20757792.83216783
2018 Monday 40369547.72141015
2018 Tuesday 32886851.119894598
2018 Wednesday 32185184.003496505
2019 Thursday 34155217.8304787
2019 Friday 37531732.720909886
2019 Saturday 14214350.677743768
2019 Sunday 20115475.77092511
2019 Monday 33411468.941382326
2019 Tuesday 39708891.32302406
2019 Wednesday 39133020.148926854
2020 Thursday 32791784.728213977
2020 Friday 32712695.004382122
2020 Saturday 14257520.796460178
2020 Sunday 17689081.461911052
2020 Monday 37542958.005249344
2020 Tuesday 35232783.71278459
2020 Wednesday 38254522.31759657
2021 Thursday 35180818.37160751
2021 Friday 31424138.257575758
2021 Saturday 17318290.874524716
2021 Sunday 18409844.696969695
2021 Monday 32794455.40796964
2021 Tuesday 35458196.96969697
2021 Wednesday 39964039.77272727
```

5. Write a program that reads in the effects of covid 19 on trade data file and prints out the average value of each type of trade (Direction column) for each weekday in the data set. The output of your program should look like:

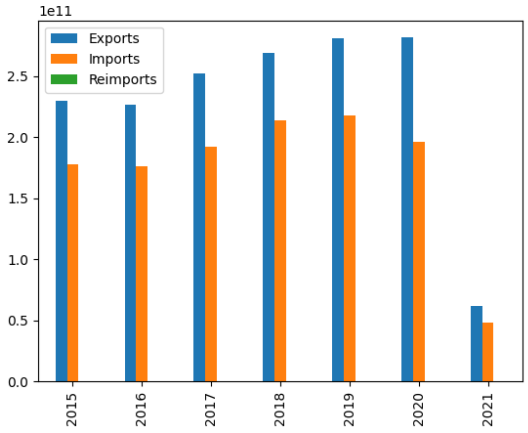
```
Exports Thursday 21277646.547561564
Exports Friday 23481516.312193245
Exports Saturday 18702610.830745343
Exports Sunday 22336156.96495801
Exports Monday 27557897.935558558
Exports Tuesday 20317972.214182343
Exports Wednesday 21180319.000581056
Imports Thursday 67582235.92042589
Imports Friday 62107692.307692304
```

```
Imports Saturday 8690909.090909092
Imports Sunday 11700615.556799104
Imports Monday 61347521.702604316
Imports Tuesday 66274482.37269166
Imports Wednesday 64652307.692307696
Reimports Thursday 726708.0745341615
Reimports Friday 690402.4767801857
Reimports Saturday 74534.16149068323
Reimports Sunday 21671.826625386995
Reimports Monday 718266.253869969
Reimports Tuesday 712074.3034055728
Reimports Wednesday 712962.9629629629
```

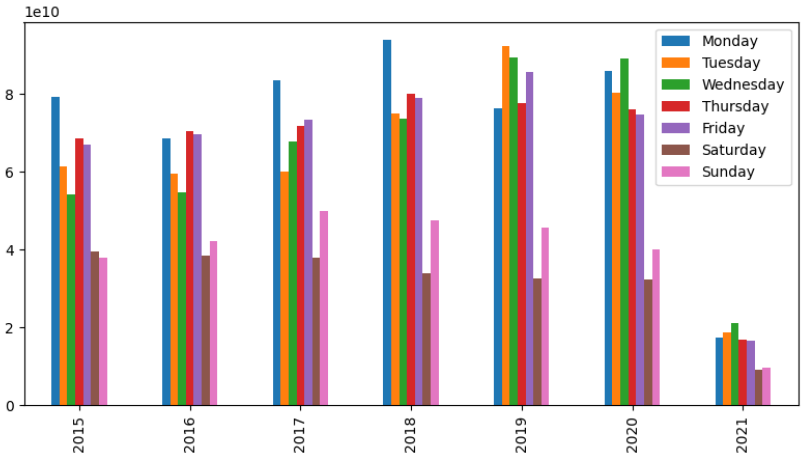
6. Write a program that reads in the effects of covid19 on trade data file, and then plots the total trade for each year in a bar graph. Your plot should look like the following:




7. Write a program that reads in the effects of covid19 on trade data file, and then plots the trade values for each direction grouped by year in a bar graph. Your plot should look like the following:



8. Write a program that reads in the effects of covid19 on trade data file, and then plots the total trade values for each year grouped by weekday in a bar graph. Your plot should look like the following:



# Lesson 20

# Visualisation

We looked at a few plot types in the previous lesson. In this lesson, we are going to focus more on doing data visualisation right. That is, what to do and what not to do. There are some very simple rules to follow that will make your visualisations information rich without misrepresenting the data. I'm going to present this lesson as a series of "what's wrong with this graph" set of questions. I'm also going to try a bit of an experiment and not specify what the answer is in the text of the book. You should be able to figure out what's wrong with one plot, because it should be fixed on the next plot, but try not to look ahead as it is worth spending some time trying to figure out what the problem is first. This sort of process will help you to sharpen up your critical thinking abilities.

I'm going to limit the text to one plot per page, to encourage you to not look ahead. For each plot, you might want to ask yourself the following questions:

1. What is the purpose of the plot? What is it trying to show?
2. What information is in the plot?
3. Can I get the information I need or want from the plot?
4. How can I make the plot better?

## 20.1 Line plots

### 20.1.1 Line Plot 1

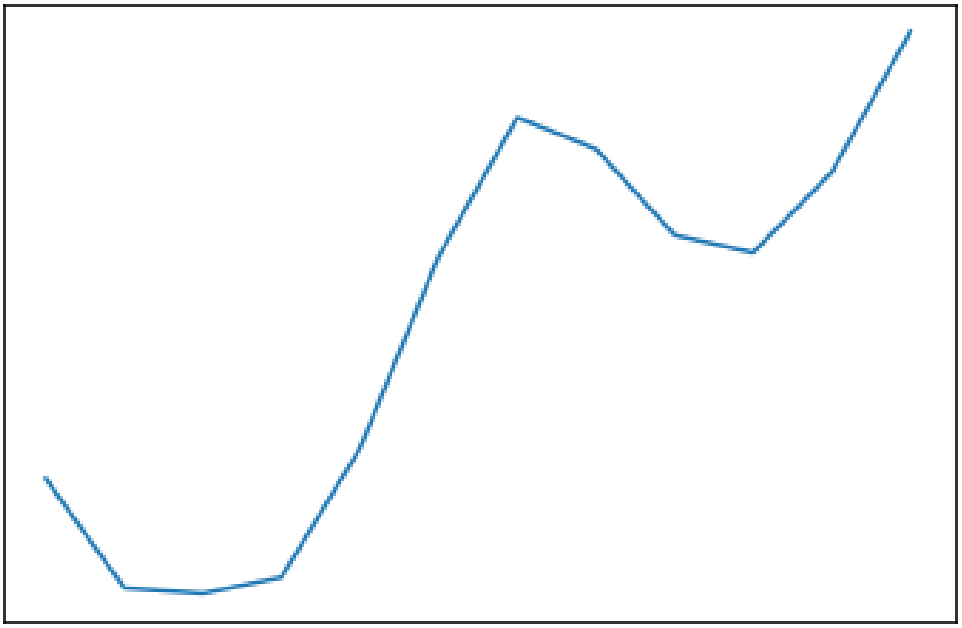


Figure 20.1: Green house gas data for the Canterbury region.

### 20.1.2 Line Plot 2

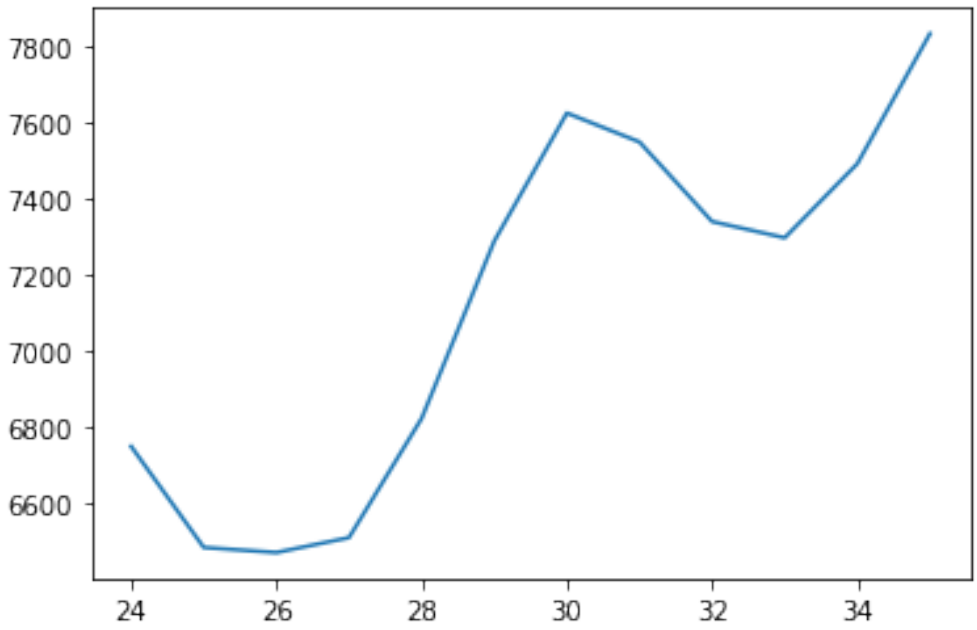



Figure 20.2: Green house gas data for the Canterbury region.

### 20.1.3 Line Plot 3

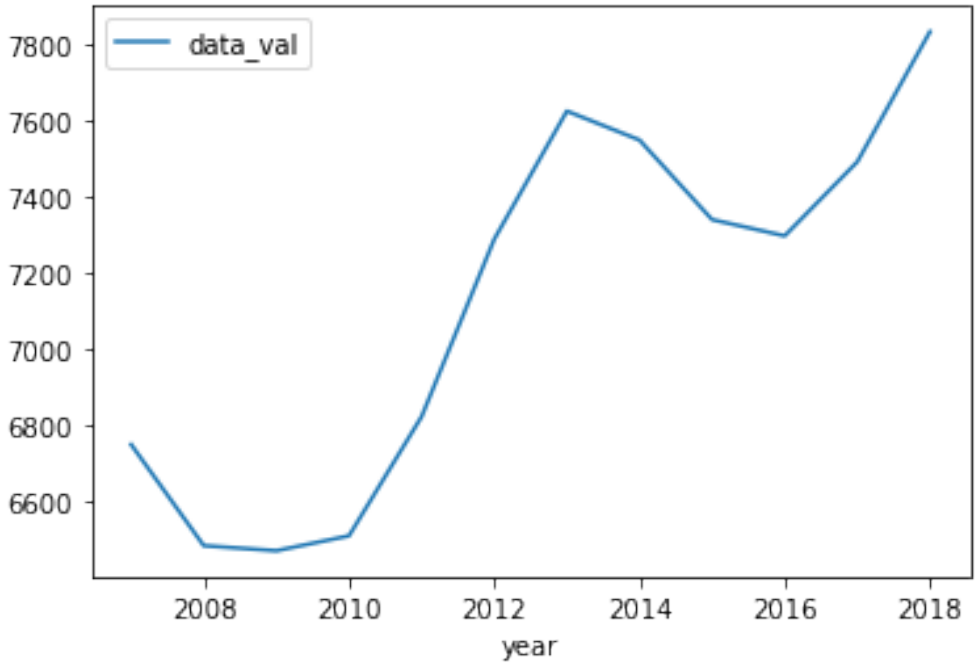



Figure 20.3: Green house gas data for the Canterbury region.

### 20.1.4 Line Plot 4

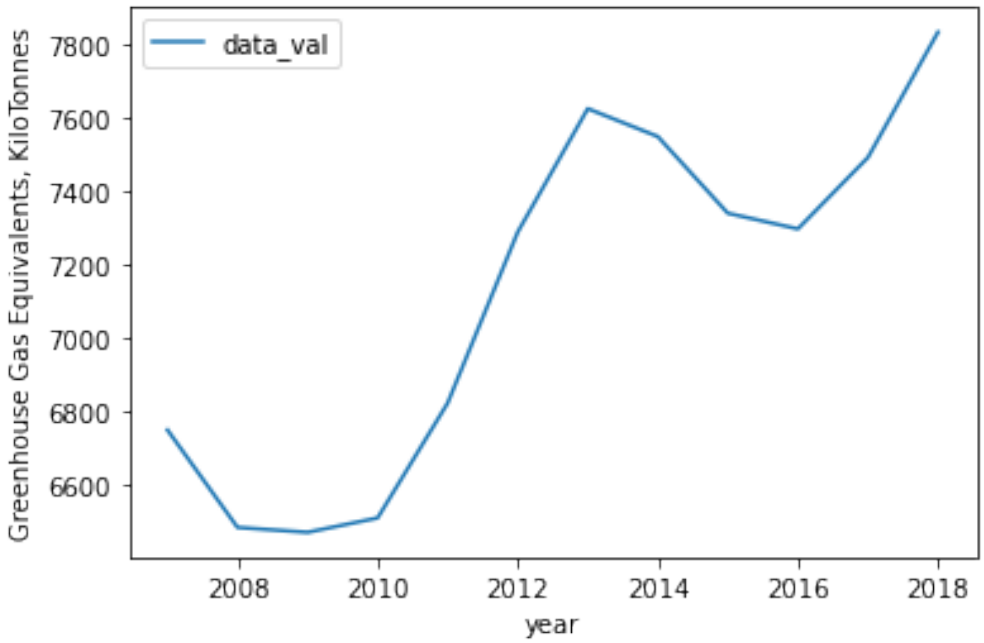



Figure 20.4: Green house gas data for the Canterbury region.

### 20.1.5 Line Plot 5

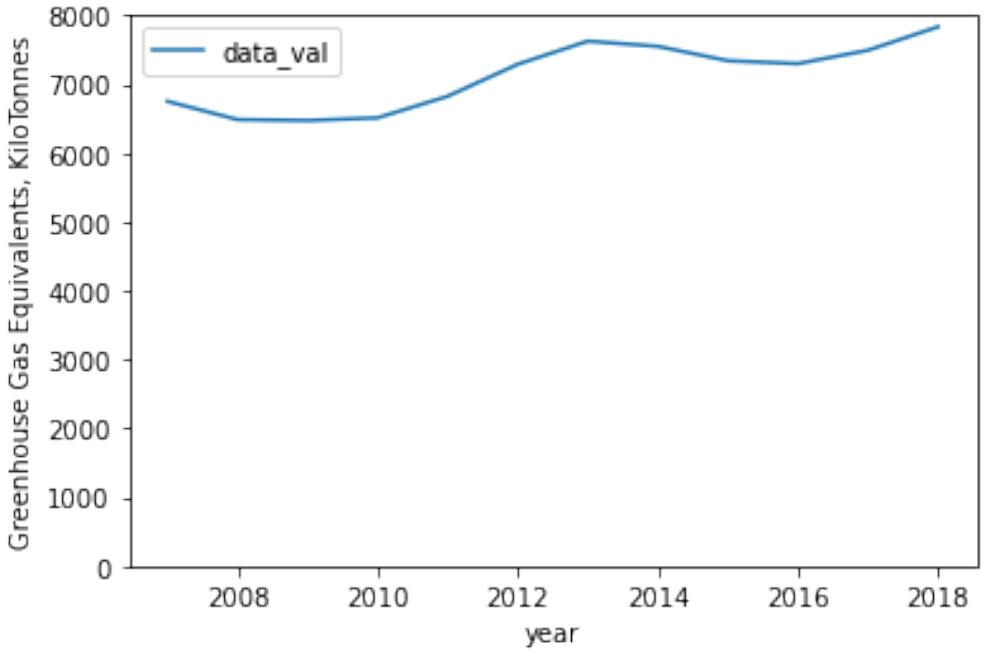



Figure 20.5: Green house gas data for the Canterbury region.

## 20.2 Bar Graphs

Generally speaking, bar graphs are most useful when there are categories along the x-axis rather than a discrete or continuous scale, although they can also be useful for discrete scales (for example, years in the previous graphs). But they can also suffer from exactly the same problems as shown in previous graphs, so we won't go through those again. However, there are problems that are particular to bar graphs that arise.

### 20.2.1 Bar Graph 1

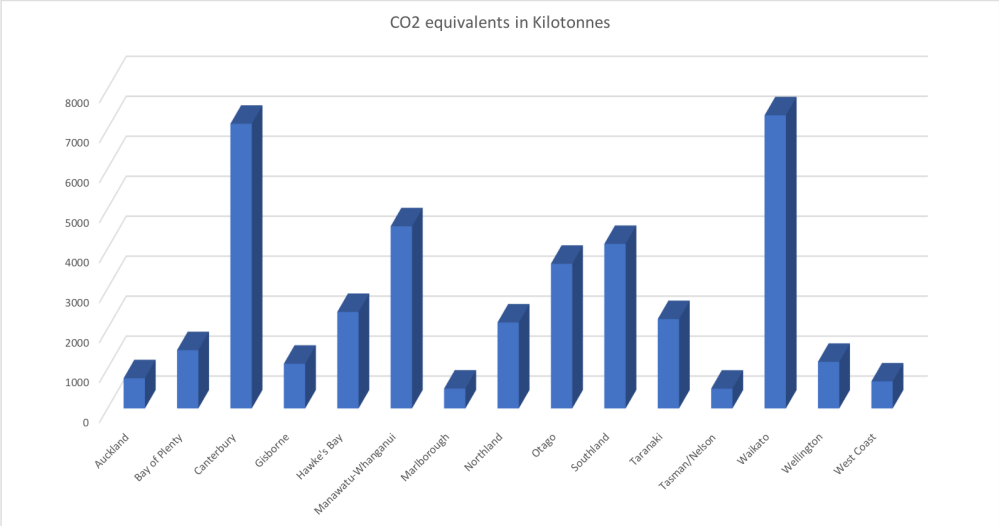



Figure 20.6: Green house gas data means for NZ.

### 20.2.2 Bar Graph 2

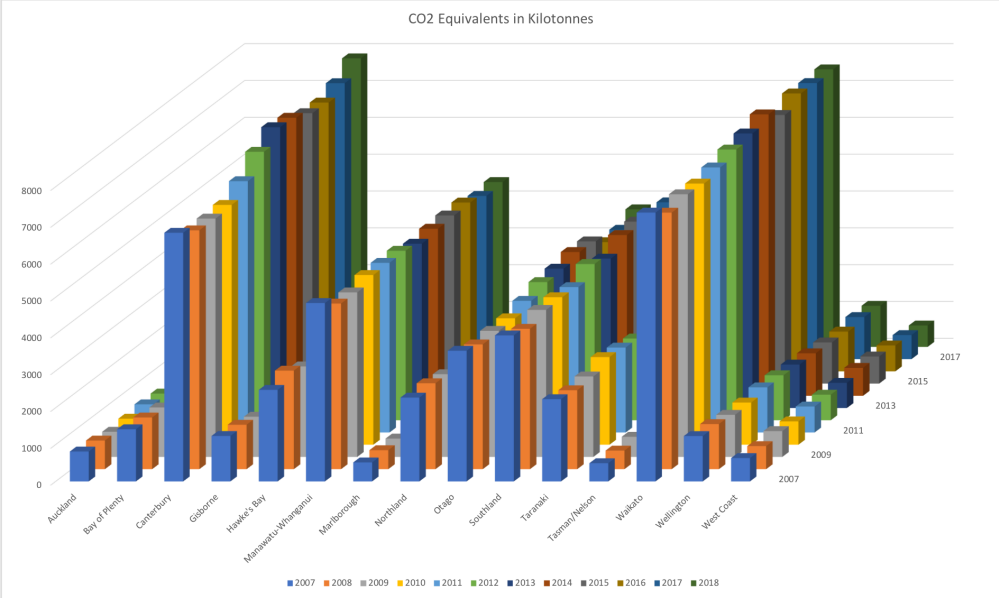



Figure 20.7: Green house gas data by region for NZ.

### 20.2.3 Bar Graph 3

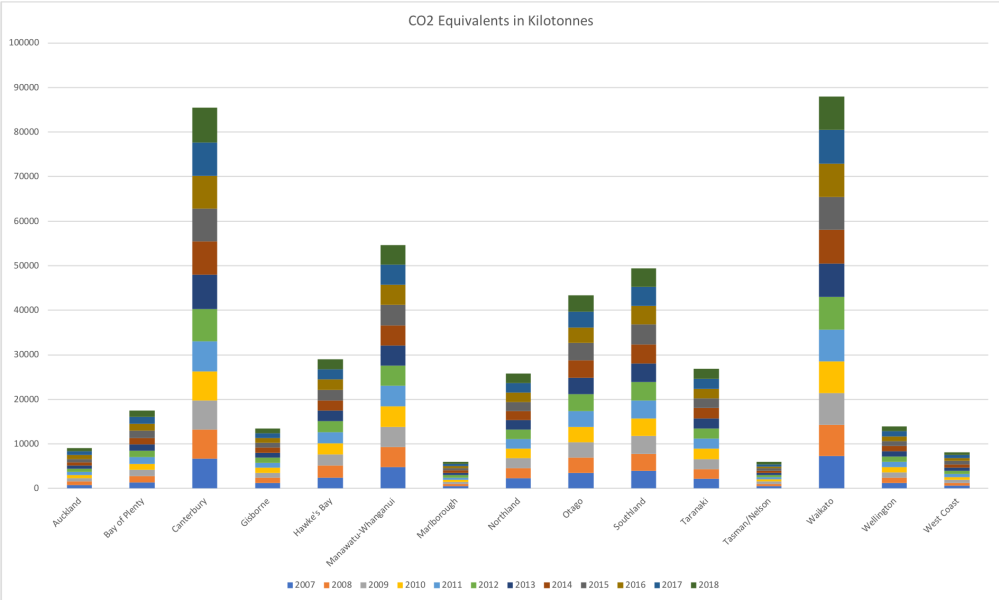



Figure 20.8: Green house gas data by region for NZ.

### 20.2.4 Bar Graph 4

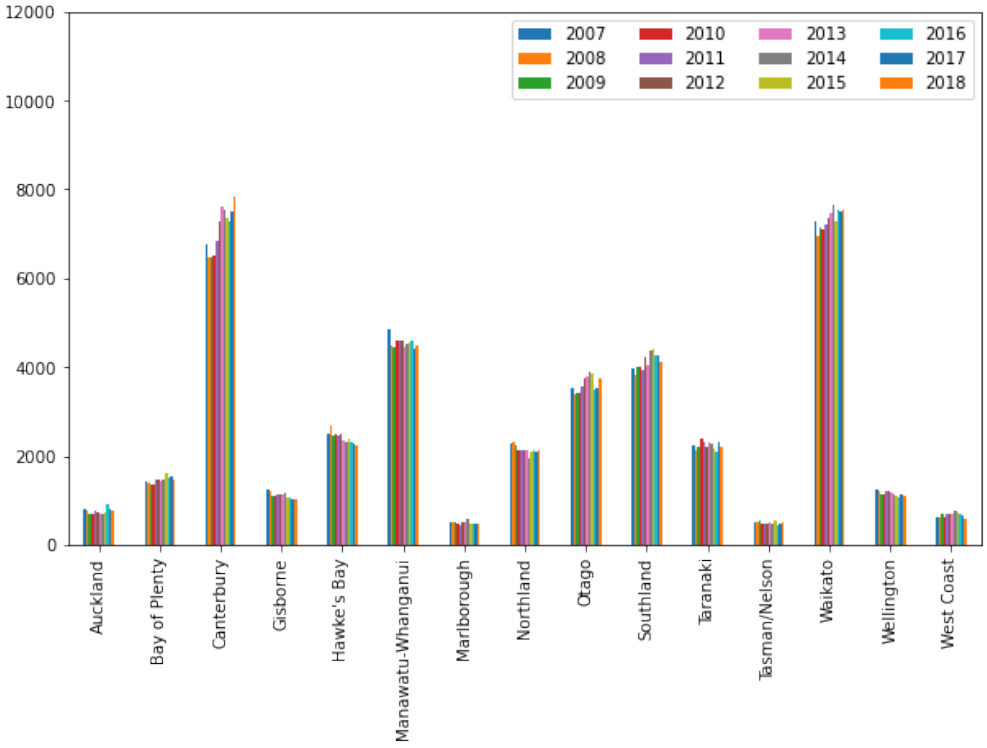



Figure 20.9: Green house gas data by region for NZ.

## 20.3 Colour

Using colour in graphs and plotting has become ubiquitous. Colour can be very useful sometimes, but care must be taken when using it. Approximately 10% of men are colour-blind, and that means using colour in your plots is going to make it very difficult for them to interpret the data. Figure 20.10 is basically uninterpretable for colour-blind people:

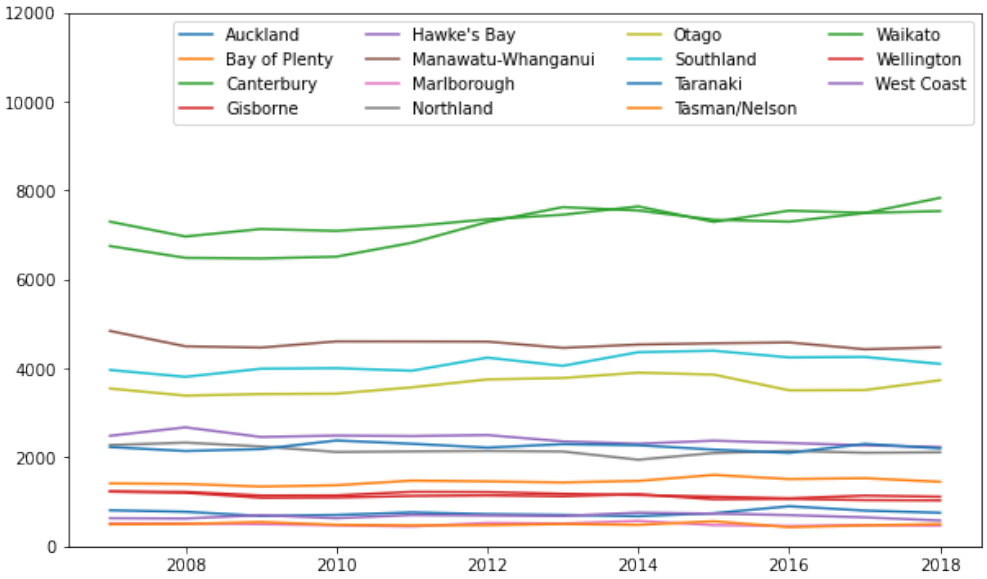



Figure 20.10: Green house gas data by region for NZ.

So how do we fix this problem? For this particular plot, there is probably no way of fixing it in it's current form. A better plot for this data is the bar plot from Figure 20.9. For the bar plot, note that the colour information is secondary and not a primary information source. It's not necessary to be able to match a colour from the legend to a colour in the plot because the information is also encoded in the position of the bars with 2007 on the left and 2018 on the right of each group of bars. Whereas for the line plot, the important information really is encoded in the colours. There are more colour blind friendly colour palettes, but even these can be confusing if there are many colours. It's best to avoid encoding essential information via colours, or at least have an alternative way of encoding that information.

Another common use of colour is in various two-dimensional representations of data, and in particular in the use of heatmaps. Heatmaps can be used for many things, but they're often good for visualising two-dimensional distribution data or two-dimensional functions. Here's a fairly simple 2D function:

$$f(x, y) = \sin(\frac{x}{7}) + \sin(\frac{y}{11}) \tag{20.1}$$

that we can put into a 2D array using the following code and visualise it using the `matplotlib` function `imshow`, where I've shifted the origin to lie in the middle of the array:

```
import numpy as np
import math
import matplotlib.pyplot as plt
mat = np.zeros([100,100])

for i in range(100):
    for j in range(100):
        x = 50-i
        y = 50-j
        mat[i,j] = math.sin(x/7) + math.sin(y/11)

plt.imshow(mat, cmap='gist_ncar')
heatmap = plt.pcolor(mat, cmap='gist_ncar')
plt.colorbar(heatmap)
plt.show()
```

Which produces the image on the left in Figure 20.11. The scale on the right of the heatmap gives the viewer

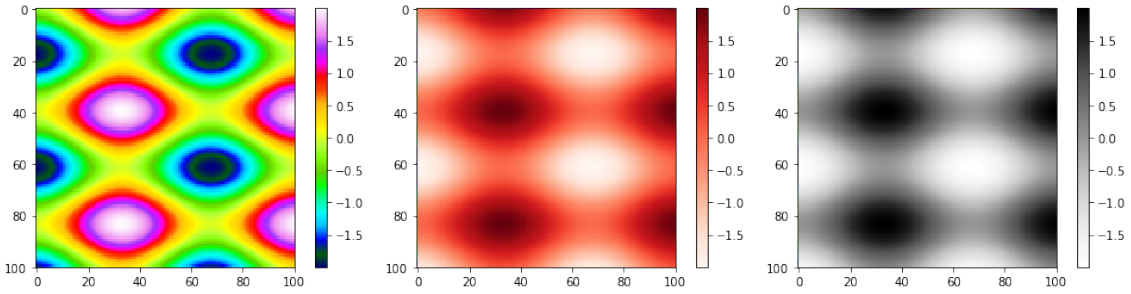


Figure 20.11: Visualising a 2D function.

an idea of the values of the function. The plot is quite pretty, but it's terrible for conveying information. I won't go into the details of colour theory here, but I have specifically chosen a bad colourmap that varies the inherent colour (the hue) to convey information. It is much better to just vary the luminance or brightness to convey this sort of information as shown in the middle. But really, the colour is not conveying any useful information at all, so you may as well stick to grey as in the image on the right.

## 20.4 Exercise

For these exercises, download the penguins size data from Blackboard. The questions assume you have loaded the data into a Pandas DataFrame.

1. For each species of penguin, create two boxplots highlighting the different weights of male and female (this is called sexual dimorphism). In one of the plots, set the scale so that the difference looks very large. In the second plot, set the scale so the difference looks very small. As a starter, the following script shows how to create a boxplot for Adelie penguins, with different boxes for males and females, with the y axis varying between 2000 and 5000:

```
import pandas as pd
import matplotlib.pyplot as plt

pengs = pd.read_csv('../Data/penguins_size.csv')

pengs[pengs['species']=='Adelie'][['body_mass_g','sex']].boxplot(by='sex')
plt.ylim(2000,5000)
plt.show()
```

2. Try to repeat the process of question 1, but now use the 'island' column instead of the 'sex' column, for the boxes.

3. Repeat the process of question 1, except now compare each of the species regardless of sex. The output plots should have three boxes, one for each species.

4. Repeat the process again but create just one plot that contains up to 6 boxes with two boxes for each species - one for male and one for female. Hint: the 'by' parameter can be a list of columns to group the plots by. You might find you get seven boxes, because for some of the Gentoos the sex is unknown.

5. If you're feeling adventurous, repeat the process again creating just one plot, but now separate out the males and females for each species for each island. In this case there will be up to 18 boxes (3 islands, 3 species, 2 sexes - $3 \times 3 \times 2$).

# Lesson 21

# High-Dimensional Visualisation

## 21.1 Introduction

If there is one message to take away from this chapter when you are dealing with data it is this: always look at your data. I cannot emphasise enough the importance of looking at your data. Data usually comes as a bunch of numbers, and while it can be useful to look directly at the numbers, it can be difficult to form a good understanding of the data from just the numbers. It is often a good idea to look at a graphical representation of the data. This is pretty easy for two- or three-dimensional data, but rather more difficult for high-dimensional data sets. For this case study, we are going to look at visualising high-dimensional data. We are going to use a simple and old data set (originally created in 1935) called the iris data set. It involves four measurements on three different types of irises. The measurements are the petal length and width, and the sepal[1] length and width, all in centimetres. The three types of irises are setosa, virginica and versicolor. There are 150 different specimens that are measured in the data set. Here are the first 5 rows:

```
[[5.1, 3.5, 1.4, 0.2],
 [4.9, 3. , 1.4, 0.2],
 [4.7, 3.2, 1.3, 0.2],
 [4.6, 3.1, 1.5, 0.2],
 [5. , 3.6, 1.4, 0.2]]
```

One can think of this data as having four dimensions. For each specimen, there are four measurements, so we can think of the data as belonging to a four-dimensional space. Directly visualising four dimensions is quite difficult (although not impossible), and it's usually much easier to visualise just two dimensions at a time. But which two dimensions should we choose? With only four dimensions, there are only a few options, but what if our dataset had 100 dimensions or 1000 or more? We are also going to look at a cancer gene expression dataset. This is a much more challenging data set to view. It has 801 samples and over 20000 dimensions. Each sample is a gene expression result from a patient who has one of 5 different types of tumour. Each dimension is an RNA-seq gene expression level to a specific gene. This is a very challenging data set to visualise.

[1] The sepals are the leaf-like parts that grow under a flower or flower bud for protection or support. See https://en.wikipedia.org/wiki/Sepal.

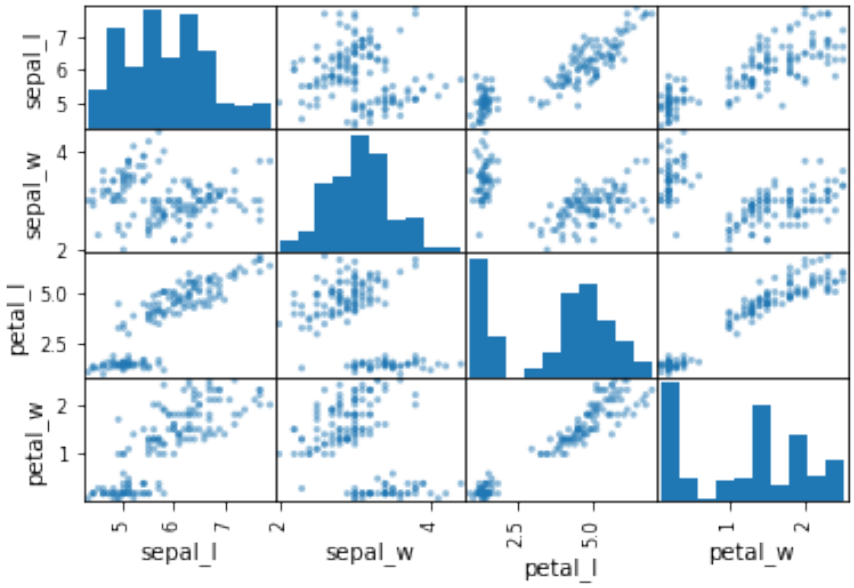



Figure 21.1: Scatter matrices for the iris data set.

## 21.2 Scatter Matrices

Possibly the easiest method is to look at the correlations between each pair of coordinates. This method works OK with a small number of dimensions, but becomes cumbersome when the number of dimensions becomes large. The scatter matrix shows scatter plots between each pair of coordinates and usually has a bar histogram of the individual dimensions on the diagonals of the figure. This makes more sense if you consider Figure 21.1. Such a figure is easy to produce in Pandas using the following code:

```
import pandas as pd
import pandas.plotting as pp
import numpy as np
import matplotlib.pyplot as plt

iris = pd.read_csv('iris_data.csv', header=None,
                   names=['sepal_l', 'sepal_w', 'petal_l', 'petal_w'])
pp.scatter_matrix(iris)
plt.show()
```

One can get some information from these plots, but they can be tricky to interpret. For example, we can easily see that the petal width and petal length are highly correlated, but the sepal length and sepal width have a much less clear relationship. Petal size appears to be related to sepal size in one group of flowers, but completely unrelated in another group. The individual histograms also tell us something useful about the coordinates. Only the sepal width appears to be normally distributed, and this would usually make me somewhat suspicious of the data. The petal width especially, looks decidedly odd and would warrant further investigation.

So this method can be a good first step, but is a little limited and can't be applied to very high-dimensional data. For example, it cannot be applied to the gene expression data set, although in Figure 21.2 I show the results on the first 50 dimensions of that data set. As can be seen, the figure is basically useless.

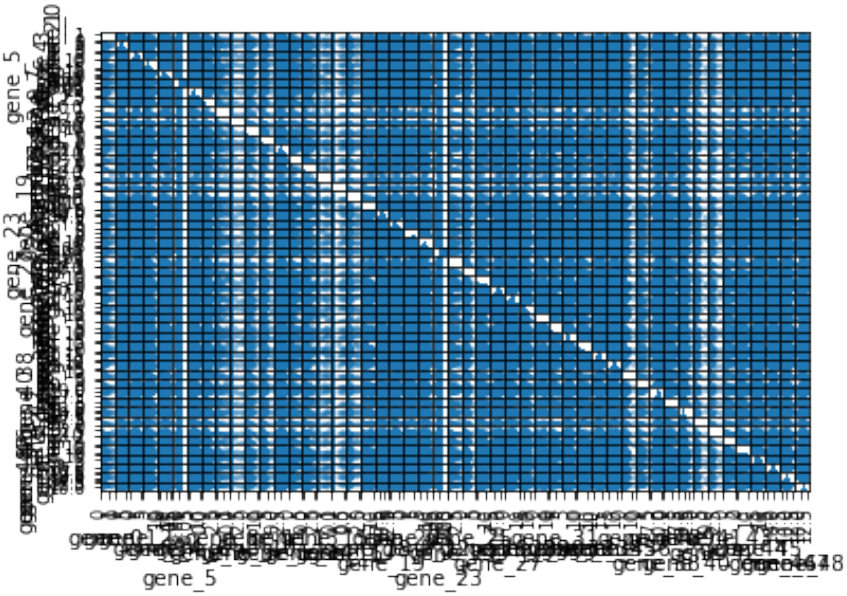



Figure 21.2: Scatter matrices for the gene data set.

## 21.3 Parallel Coordinates

Another relatively simple visualisation method is called parallel coordinates. In this method we treat each sample as a jointed line. Each dimensional value is treated as a 2D point with coordinates (val, dimension number). So for example, taking the first row of the iris data set, we would plot lines that join the points $(1, 5.1), (2, 3.5), (3, 1.4), (4, 0.2)$. This is shown in Figure 21.3 which was generated using the following code:

```
import pandas as pd
import pandas.plotting as pp
import numpy as np
import matplotlib

labels = pd.read_csv('iris_labels.csv', header=None, names=['type'])
iris = pd.read_csv('iris_data.csv', header=None,
                   names=['sepal_l', 'sepal_w', 'petal_l', 'petal_w'])
iris_all = pd.concat([labels, iris], axis=1)
pp.parallel_coordinates(iris_all, 'type')
plt.show()
```

One can clearly see from Figure 21.3 that the different iris types tend to have differing measurements and do cluster according to label, although with some overlap.

Although better at handling very high dimensional data, this sort of plot is not very good at handling data with many samples. Even a relatively small data set such as the gene sequence data set (many dimensions, but only 800 samples), produces unreadable plots. In this case, I go back to just using Numpy and matplotlib, but even then, the plot is unreadable as shown in Figure 21.4. The code to generate the plot is:

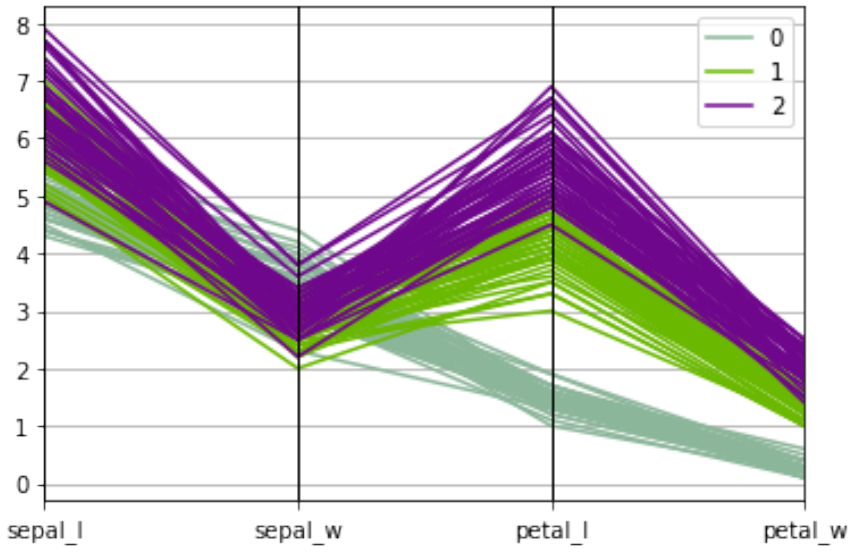



Figure 21.3: Parallel coordinates for the iris data set.

```
import numpy as np
import matplotlib
gene_np = np.genfromtxt("gene_data.csv", delimiter=",")
gene_np = gene_np[1:,1:] # need to remove first row and first column
plt.plot(gene_np.T) # take the transpose to generate plot
plt.show()
```

## 21.4 Dimensionality Reduction

For data visualisation, using two or three-dimensions for the visualisation is ideal, with 2D being the simpler. However, the methods we've seen so far are very limited in either the number of dimensions or the number of samples. We really would like to reduce the number of dimensions in our data down to 2 or 3, yet still maintain the maximum information we can in any resulting plot.

Let's think of a simpler example. Imagine you want to draw a three-dimensional object such as a pencil. The drawing is a two-dimensional representation of a three-dimensional object, so we're effectively reducing the dimensions of the pencil. From which angle should you draw the pencil? It depends on what you're trying to achieve, but generally you should choose a direction that maximises the information content of the drawing. For an object like a pencil, that is probably a drawing that shows its long axis, rather than a drawing of a circle (looking at the pencil from end on). This is the main idea of principal components analysis (PCA) - it tries to find a direction to draw the data that maximises the spread of that data.

### 21.4.1 Principal Components Analysis

Principal components analysis (PCA) gives us one way of visualising data in a methodical and automatic fashion. It does not choose two of the input dimensions, but rather reorients the data so that the axes are oriented according to how spread out the data is - the largest spread along the first dimension, second largest spread along the second dimension etc. These new dimensions need not align with the original

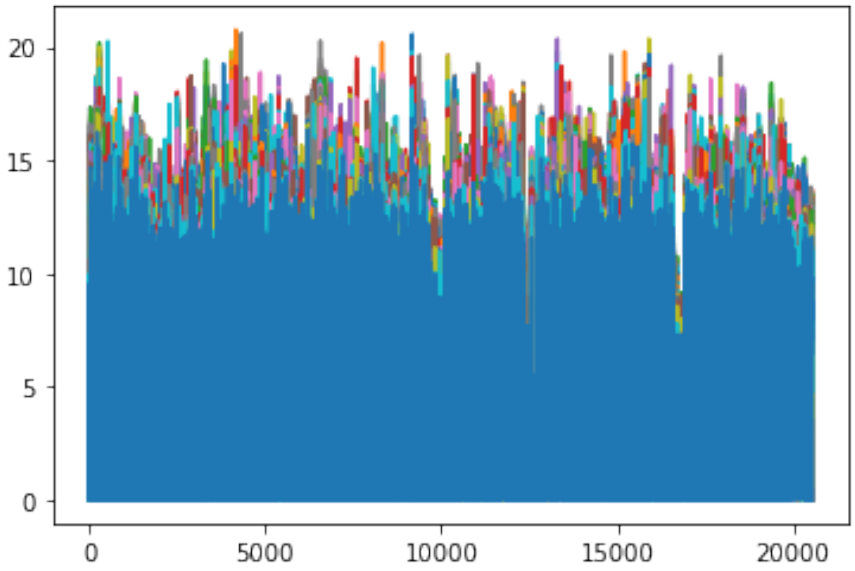



Figure 21.4: Parallel coordinates for the gene data set.

data, but are some combination of the original dimensions. Figure 21.5 gives a basic idea of how data are transformed in the case of two-dimensional input data (we wouldn't usually apply principal components in this case, but it is useful for demonstration purposes). The original data is spread out on a diagonal, but the PCA transformed data shows the major dimension spread along the x-dimension, and the smaller spread along the y-dimension. The beauty of PCA is that we can apply it to any dimensional data as we will see.

PCA is a common data visualisation and pre-processing step, but I will not describe why or how PCA works - but I will operationally run through how to use it, how to interpret the results, and how to do quick visualisations. PCA is an example of a linear method of dimensionality reduction. All linear methods try to find a simple projection that maximises some criteria. In the case of PCA, it tries to find a projection that maximises the spread of the data. This works pretty well for unlabelled data. For labelled data, there are more options, such as maximising the distance between classes, but the basic idea remains. The important thing to take from this lesson is that you can project data from high dimensions into fewer dimensions without losing too much information (and this idea can also be used for compressing data).

We are going to use a new package for applying PCA called scikit-learn[2]. First we load the data, then run PCA:

```
import numpy as np
from sklearn.decomposition import PCA

iris_data = np.genfromtxt('iris_data.csv', delimiter=',')
iris_labels = np.genfromtxt('iris_labels.csv', delimiter=',')

pca = PCA()
pca.fit(iris_data)
```

Note that sklearn uses a standard way of doing things that involve instantiating a variable based on the type of processing we want (in this case PCA), and then calling the variable's fit method using the data we

[2]Don't forget to install the module in thonny if you haven't already done so.

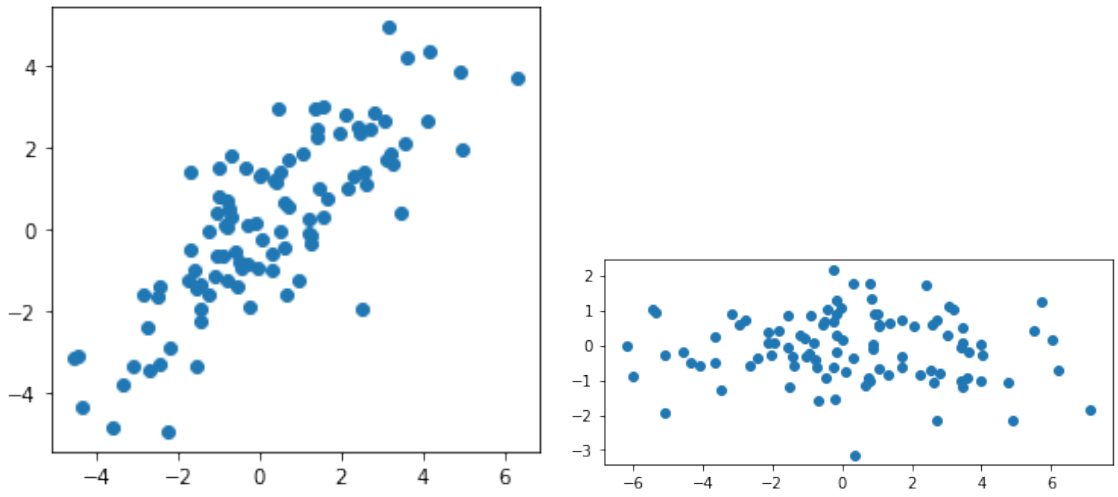



Figure 21.5: The result of PCA on two-dimensional data. On the left is the original image with the main spreading direction along the diagonal. On the right is the PCA-transformed data where the main spreading direction is along the x-axis and the minor spreading direction is along the y-axis.

want to process. After we've done that, there are a few things we can look at. Since PCA rotates the data so the largest spread is along the first dimension, the variances along those dimensions become meaningful. We can print them out:

```
>>> print(pca.explained_variance_)
array([4.22824171, 0.24267075, 0.0782095 , 0.02383509])
```

And we see that the amount of spread in the first dimension is much higher than the second which is much higher than the third etc. In fact, it is likely that the spread in the third and fourth dimensions is probably mostly noise, so if we wanted to compress our data, simply chopping off those dimensions would be a good start. In any case, it is pretty easy to visualise the first two dimensions as a scatter plot, but we need to rerun the fit with only two components:

```
pca = PCA(n_components=2)
pca.fit(iris_data)
projected_iris = pca.transform(iris_data)
fig,ax = plt.subplots()
ax.scatter(projected_iris[:,0],projected_iris[:,1], c=iris_labels)
ax.set_aspect('equal')
```

Which yields Figure 21.6. The colours of the points show quite a clear separation of the different iris classes so that's good news for the measurements we've made indicating that they are good discriminators for the classes we have.

But again, this is only for a low-dimensional data set. What happens when we try it on the gene data? Again, as a first step, we should get an idea of the how spread out the data is, so we use PCA without limiting the number of components:

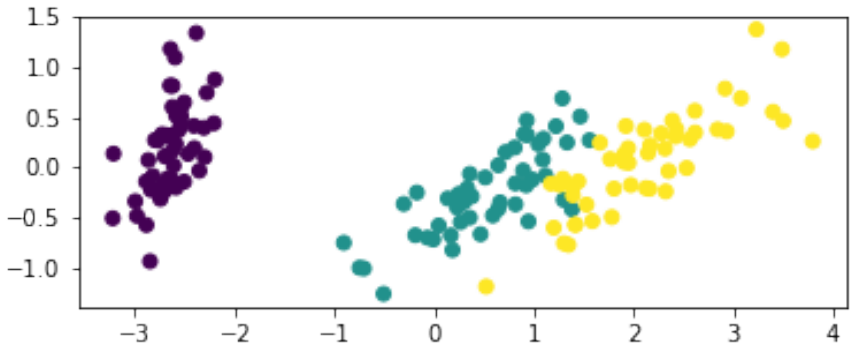



Figure 21.6: PCA for the iris data set.

```
gene_data = np.genfromtxt("gene_data.csv", delimiter=",")
gene_data = gene_data[1:,1:]
pca = PCA()
pca.fit(gene_data)
print(pca.explained_variance_[:10])
```

Which prints out the variance along the first 10 components:

```
[5736.64615291 3804.48343861 3430.79249442 2354.55948193 1309.55926182
 1076.68560553  962.40410007  566.01350645  509.63512275  444.36754592]
```

This isn't great news for visualising. Basically it's saying that the first component has a variance of 5700 (absolute values aren't important), and the tenth component has a variance of 444. The first component explains ten-times as much variance as the tenth. But the third component explains more than half as compared to the first, so if we look at only the first two components, we are going to be missing out on a lot of information. But it's still worth looking at a visualisation. Figure 21.7 shows the results of plotting the first two components of the gene data. Each different colour represents a different type of tumour. Somewhat surprisingly, reducing the data from 20000 dimensions down to 2 has actually maintained quite a bit of separation between the different types of tumour. If I wanted to build an automatic classifier based on this data, I suspect we could get away with as few as 10 dimensions.

### 21.4.2 Non-linear Dimensionality Reduction

Finally, I want to show a quick example of non-linear dimensionality reduction. There are several methods which include: Kernel PCA, Locally Linear Embedding, Isomap, Multi-dimensional scaling, and t-SNE. All of these methods are available in scikit-learn, with many of them in the `sklearn.manifold` module. Basically, linear dimensionality reduction can be accomplished using matrix multiplication, whereas non-linear reduction cannot be. To give a very simple example for a single variable, $x$, a linear transformation is always of the form $ax+b$, where $a$ and $b$ are some constants. For example, the transformation $x' = 3x+1$. A non-linear transformation is more complicated - it can use powers, exponentiation, trigonometric functions, logarithmic functions etc. Simple example of a non-linear transformations include: $x' = 2x^2 + 7$; $x' = 3^x$; $x' = \sin(x)$. The same idea works on high dimensional data. For example, given some three-dimensional

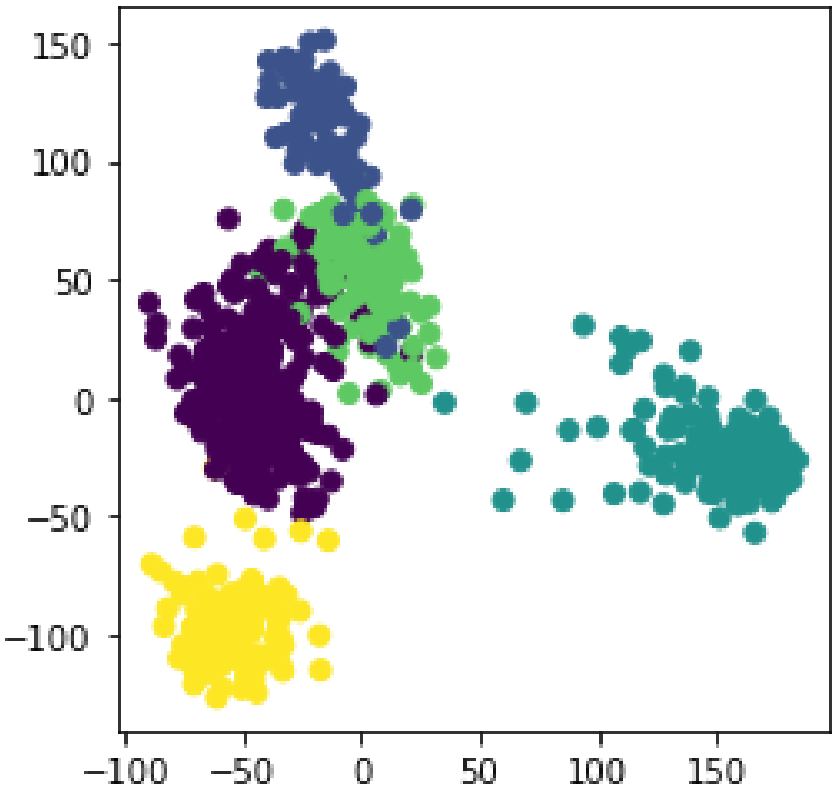



Figure 21.7: PCA for the gene data set.

data with coordinates $(x, y, z)$, a linear transformation into two-dimensions always looks like:

$$x' = ax + by + cz + d$$
$$y' = ex + fy + gz + h$$

where $a, b, c, ...$ are all constants. A non-linear transformation is much more general. Here's a very simple example:

$$x' = xy + yz + xz$$
$$y' = xyz$$

For linear transformations the problem is fairly straightforward - we know exactly what the form of the transformation is and we just have to decide how to find the best transformation. But for non-linear transformations, how do we decide what the best form of the transformation is, let alone the best specific transformation of that type? There is no good answer - there are an infinite set of non-linear transformations we could choose, but there are some that generally work better than others. Possibly one of the best is called t-SNE, and scikit-learn includes t-SNE in its set of transformations. Here is how to use it:

```
from sklearn.manifold import *
tsne = TSNE(n_components=2)
projected_gene = tsne.fit_transform(gene_data)
fig,ax = plt.subplots()
ax.scatter(projected_gene[:,0],projected_gene[:,1], c=int_gene_labels)
ax.set_aspect('equal')
```

Note that the interface is a little bit different (`fit_transform`, rather than `fit` followed by `transform`). The results are remarkable and are shown in Figure 21.8. The figure shows clear separation between almost all of the different tumour clusters which is much better than PCA. Remember that these methods are based on analysing the data directly - they have no knowledge of the class the data comes from, but t-SNE shows that there are clear differences between the gene expressions of individuals with different tumour types in almost all cases.

## 21.5 Summary

We've looked at four different ways of visualising high-dimensional data. Here is a summary of what we've learnt:

1. If there are relatively few dimensions (up to 10 or 20) then scatter matrices can be informative.
2. If there are relatively few samples, then parallel coordinates are worth trying.
3. For all cases, it is worth trying dimensionality reduction, first try PCA as it is one of the simplest methods that often gives good results.
4. If PCA does not give good enough results, try a non-linear method such as t-SNE.

There are many other methods, but these methods are a good place to start.

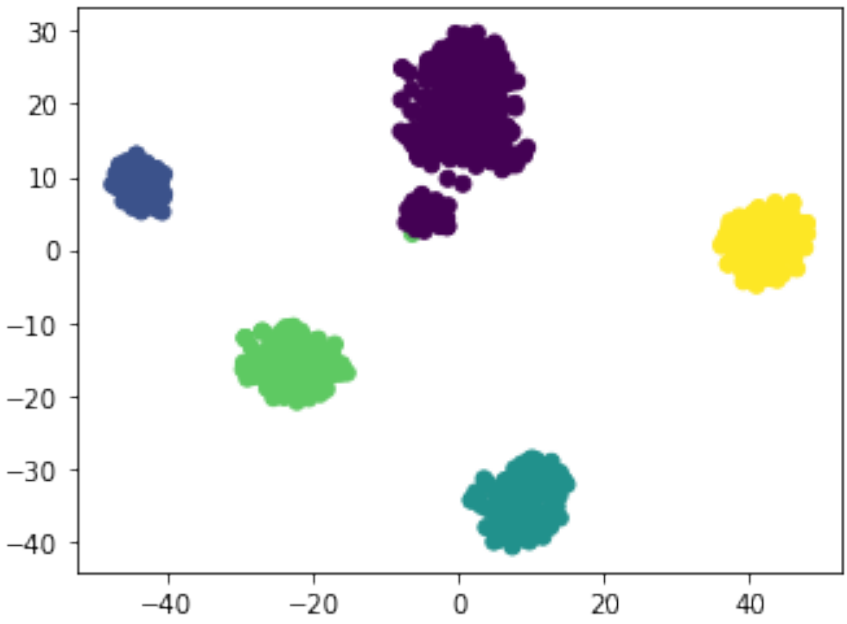



Figure 21.8: t-SNE for the gene data set.

## 21.6 Exercises

1. Download the files "iris_data.csv" and "iris_labels.csv". Write code to load the data into python.Then try each of the methods introduced in this chapter on that data.

2. The labels in "iris_labels.csv" are integers which is convenient for the methods introduced in this chapter. But the labels in "gene_labels.csv" are names indicating what type of tumour the patient has. Unfortunately, we can't colour the dots with a name, but we can use integers. Download the "gene_data.zip" file, then load in both the data and the labels. Write python code to create a labels list with integer entries. For example, the first 5 lines of "gene_labels.csv" are:

```
,Class
sample_0,PRAD
sample_1,LUAD
sample_2,PRAD
sample_3,PRAD
sample_4,BRCA
```

   But the desired integer label values are: `[0,1,0,0,2]`.

3. Try PCA and t-SNE on the gene data and reproduce the results in this chapter.

4. Look at the page: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.manifold. Choose two of the other methods such as Isomap or MDS, and try those methods on both the iris data and the gene data.

5. Do a dimensionality reduction on the Tara Hills temperature data set (also available on Blackboard). You will have to do some data clean up first, and you should try two different types of cleaning:

   (a) Ignore missing values: to read in the data and only use the years without missing values, you can use the following code:

```
read_data = np.genfromtxt('Tara_Hills_temp.csv', delimiter=',', skip_header=1,
                          missing_values='NA')
read_data = read_data[~np.isnan(data).any(axis=1)]
years = read_data[:,0]
data = read_data[:,1:13]
```

You should then be able to apply PCA or t-SNE on the data matrix as above.

(b) Replace missing values by the monthly mean. For this you will need to compute the mean for each month ignoring missing values (use `np.nanmean`), then go through the matrix and replace any `nan`[3]s with the mean. There are a few ways of doing this, but the easiest is to go through each column of the data matrix (use a for loop), and then go through each row of a column (another for loop), and if there's a `nan` replace it with the column mean.

[3] nan means Not a Number

# Lesson 22

# Random Numbers and Statistical Tests

Statistical testing forms the basis for deciding the results of experiments in almost all areas of science. For example, in the health sciences, we might want to know if a given drug improves outcomes for patients with a certain condition. In neuroscience, we might want to know if exercise or reading or watching TV helps increase the activity in certain parts of the brain. In zoology we might want to know what the effect of planting native flora in suburban gardens has on native fauna. You get the idea. A lot of statistical testing involves testing whether two or more groups are the same or different along some measurement. Of course, you can study statistics for a lifetime and still not know all of it, so I am not even going to try and give anything like an overview in one lecture. Instead, I am going to focus on a simple, yet powerful technique that can be applied in many situations and takes particular advantage of computational power. It also has the advantage of requiring fairly little statistical knowledge, and is a good way to exercise our programming capabilities. It's called the bootstrap, and it relies on random numbers, so we are going to discuss random numbers first.

## 22.1 Random Numbers

Generating random numbers is important in many parts of computing, and especially important in data heavy fields such as data science, artificial intelligence and statistics. Generating random numbers is quite difficult in general and impossible for deterministic procedures like computer programs. Real random numbers can be generated from physical systems such as nuclear decay events, or thermal fluctuations of a physical device, but most random numbers used in computer programs are pseudo-random numbers[1]. Pseudo-random number generators are initialised using a seed, and if the same seed is used for initialisation, then the same sequence of numbers are generated. So, although they look random, they aren't really. Python has a standard library called `random` that has various random number functions, but we are going to use the numpy version. First we need to create a random number generator object:

```
import numpy.random as rand

rng = rand.default_rng()
```

The function `default_rng` takes a seed as a parameter, but if no seed is given, then some unpredictable seed will be sourced from the operating system. Depending on the machine and operating system, this unpredictable seed might be a true random number (but you shouldn't rely on that). If you want to be able

[1] An example of physical real random numbers is provided in Intel chips: https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

to generate the same sequence of numbers again, then passing in an explicit seed will initialise the random number generator into the same state.

To actually generate random numbers, we need to decide what sort of numbers we want. That is, what is the distribution. Common distributions include the uniform distribution (all numbers equally likely) and the normal distribution (the famous bell curve). A distribution function or probability distribution function tells us how probable it is that a particular number will be generated. We can look at sampled distributions by plotting a histogram of all the numbers in the sample. The following code generates 1000 numbers from a uniform distribution and plots the histogram shown in Figure 22.1 (which should be flat):

```
import numpy.random as rand
import matplotlib.pyplot as plt

rng = rand.default_rng()
rnums = rng.uniform(size=1000)
plt.hist(rnums, bins=10)
plt.show()
```

If we replace `rng.uniform(size=1000)` with `rng.normal(size=1000)`, then the resulting histogram is as seen on the right. Note that the `bins` optional parameter specifies how many bins to create and will divide the range equally between the highest and lowest result. The first bin goes from 0 to 0.1 approximately, and the height of the bar shows how many numbers appear in that range in the `rnums` array.

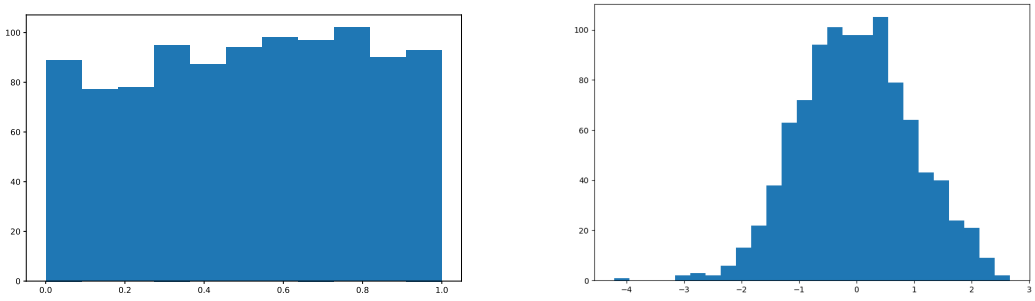



Figure 22.1: Histograms of random uniform numbers on the left and random normal numbers on the right.

If we increase the number of bins, the histogram becomes noisier, but if we increase the number of random numbers, the histogram becomes smoother and approaches the underlying distribution function as the number of samples approaches infinity. In Figure 22.2 the top row shows 100 bins and 1000 samples, the bottom row shows 100 bins and 1000000 samples.

One can also generate random integers using the `rng.integers` function. In this case we specify the lower and upper values of the range, and can either include the upper endpoint or not in the numbers generated. So, for example, to simulate the rolling of a fair dice 1000 times, we would use the following code:

```
dicenums = rng.integers(low=1, high=6, size=1000, endpoint=True)
```

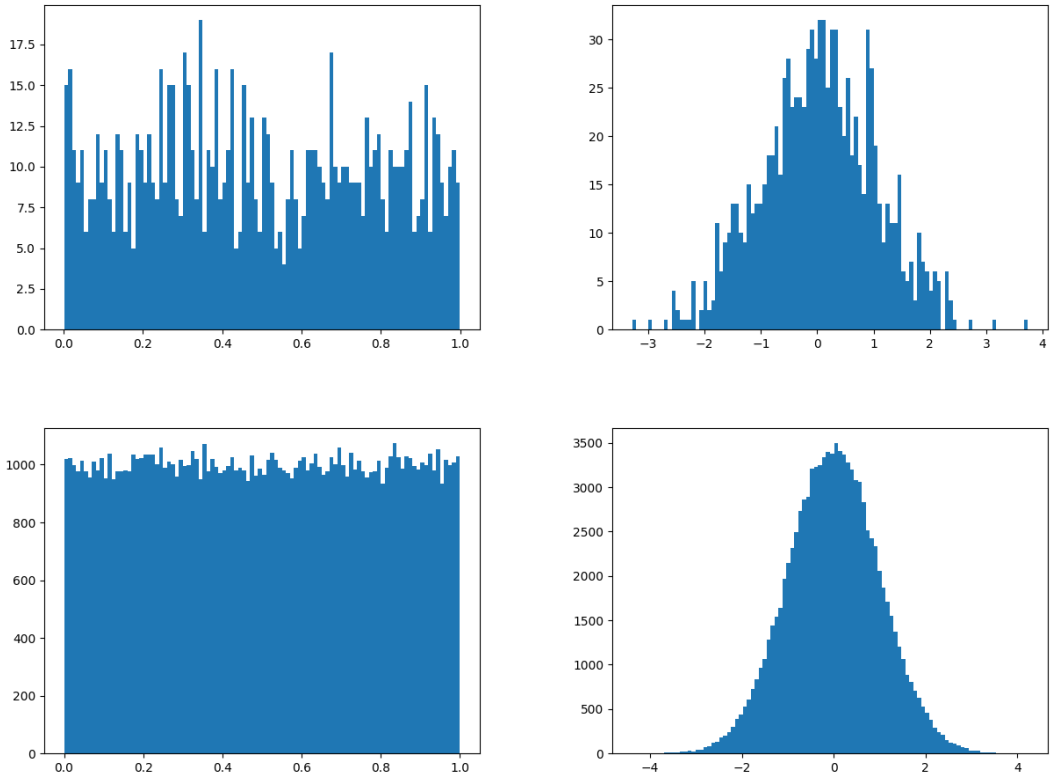


Figure 22.2: 1000 samples and 100 bins on the top row. 100000 samples and 100 bins on the bottom row.

## 22.2 The Bootstrap

### 22.2.1 Basic Bootstrap

The basic idea of the bootstrap is best described with an example. Let's assume we have loaded in the penguins data set and are interested in the mean weight of Adelie penguins. We can simply compute that directly from the data. But what if we also want to know how confident we should be that the computed mean is reliable? The bootstrap has a blindingly simple solution to the problem, and the algorithm looks like this (assume the ):

```
data = pengs[pengs['species']=='Adelie']['body_mass_g']
do the following some number of times (say 1000):
    let ndata be a random sample taken from data with replacement
    compute the mean of the new data set ndata
    add the mean to a list of computed means
now inspect the distribution of means
```

The distribution of computed means tells us how reliable the initially computed mean is. If the distribution is narrow, then we are very confident, but if the distribution is wide, then we would be less confident. Here is an actual python function for computing this distribution:

```
import numpy.random as rand
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def bootstrap_mean(data, num_samples):
    rng = rand.default_rng()

    mean_list = []
    for i in range(num_samples):
        new_data = rng.choice(data, size=len(data), replace=True)
        mean = np.mean(new_data)
        mean_list += [mean]

    return np.array(mean_list)


pengs = pd.read_csv('../Data/penguins_size.csv')
means = bootstrap_mean(pengs[pengs['species']=='Adelie']['body_mass_g'], 10000)
plt.hist(means, bins='auto')
plt.show()
```

And the result is shown in Figure 22.3. On the left, I've used all data points in the original sample, but on the right, I've only used 10 data points. We can see that the range of possible mean values is larger on the right indicating that we should have less confidence in the sample mean.

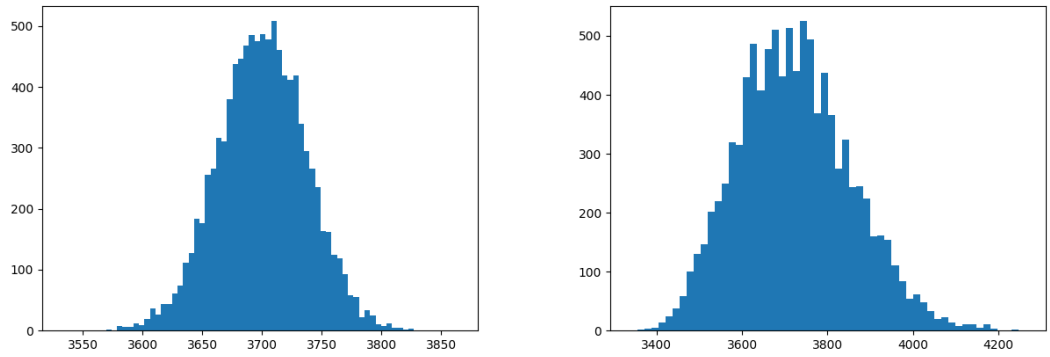



Figure 22.3: Left figure: a distribution of the mean mass of Adelie penguins, using 10000 resamples with 150 data points per sample. Right figure: the same computation, but this time starting with only 10 data points.

It's also usual and useful to be able to specify a confidence interval for any given computed statistic. The histogram gives us a good qualitative idea of how accurate our estimate of the mean is, but we often want something more quantitative. For the bootstrap, computing a confidence interval is fairly trivial. If we want the 95% confidence interval for example, we want to ignore the smallest 2.5% of computed means, and the largest 2.5% of computed means, leaving 95% of the computed means in the middle. To do that in Python is very easy:

```
>>> np.percentile(means, [2.5, 97.5])
array([3627.81456954, 3773.67549669])
```

The result being that we can say we are 95% confident that the mean mass of an Adelie penguin is between 3563g and 3615g.

### 22.2.2 Comparing two populations

We often want to ask questions regarding whether two populations are the same or different, or indeed how different. If we can measure the whole populations, then it is easy to answer these questions, because we can compute differences exactly. But usually we can't measure the whole population. For example, we might want to know about two related questions:

1. Are male Adelie penguins heavier than female Adelie penguins?

2. What is the difference in the mean mass of male and female Adelie penguins?

Since we only have a sample of the population, we can't answer those questions directly. Instead, we can answer related questions:

1. If our two samples were sampled from the same underlying population, what is the probability that we find by chance that male penguins are heavier than female penguins as observed? and

2. If our two samples were sampled from the same underlying population, what is the probability that the observed difference in means occurred by chance?

This is a subtle but important distinction. We can answer these last two questions with the bootstrap.

In both cases, similar bootstrap procedures are used, but what is computed is slightly different. In the first, for each bootstrap sample, we simply ask if the mean of the males is greater than the mean of the females, and count the number of times this occurs compared to the total number of bootstrap samples as in the following:

```
import numpy.random as rand
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def bootstrap_mean_greater(data1, data2, num_samples):
    rng = rand.default_rng()

    mean_list = []
    for i in range(num_samples):
        new_data1 = rng.choice(data1, size=len(data1), replace=True)
        new_data2 = rng.choice(data2, size=len(data2), replace=True)
        mean1 = np.mean(new_data1)
        mean2 = np.mean(new_data2)
        mean_list += [mean1>mean2]

    return 1.0-mean_list.count(True)/len(mean_list)

pengs = pd.read_csv('../Data/penguins_size.csv')
males = pengs[(pengs['species']=='Adelie') & (pengs['sex']=='MALE')]['body_mass_g']
females = pengs[(pengs['species']=='Adelie') & (pengs['sex']=='FEMALE')]['body_mass_g']
males_bigger_chance = bootstrap_mean_greater(males, females, 10000)
print('Probability that observed difference was due to chance = ', males_bigger_chance)
```

Which produces a probability of 0. Note that even though the probability as reported is 0, it is not exactly so. We took 10000 bootstrap samples, so what we can actually say is the probability of the result occurring by chance is less than $1/10000$.

In the second, we just compute the difference of the means and produce a new distribution of differences from which we can plot a histogram or produce a confidence interval:

```
import numpy.random as rand
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

def bootstrap_mean(data, num_samples):
    rng = rand.default_rng()

    mean_list = []
    for i in range(num_samples):
        new_data = rng.choice(data, size=len(data), replace=True)
        mean = np.mean(new_data)
        mean_list += [mean]

    return np.array(mean_list)

def bootstrap_mean_difference(data1, data2, num_samples):
    rng = rand.default_rng()

    mean_list = []
    for i in range(num_samples):
        new_data1 = rng.choice(data1, size=len(data1), replace=True)
        new_data2 = rng.choice(data2, size=len(data2), replace=True)
        mean1 = np.mean(new_data1)
        mean2 = np.mean(new_data2)
        mean_list += [mean1-mean2]

    return np.array(mean_list)

pengs = pd.read_csv('../Data/penguins_size.csv')
males = pengs[(pengs['species']=='Adelie') & (pengs['sex']=='MALE')]['body_mass_g']
females = pengs[(pengs['species']=='Adelie') & (pengs['sex']=='FEMALE')]['body_mass_g']
mean_diff = bootstrap_mean_difference(males, females, 100000)
confidence_interval = np.percentile(mean_diff, [2.5, 97.5])
print('95% Confidence interval of diff is: ', confidence_interval)
plt.hist(mean_diff, bins=100)
plt.show()
```

Which produces the histogram in Figure 22.4 and the following output:

```
95% Confidence interval of diff is:  [575.68493151 775.        ]
```

From the answers to both questions, it would be reasonable to conclude that male Adelie penguins are heavier than female Adelie penguins almost certainly.

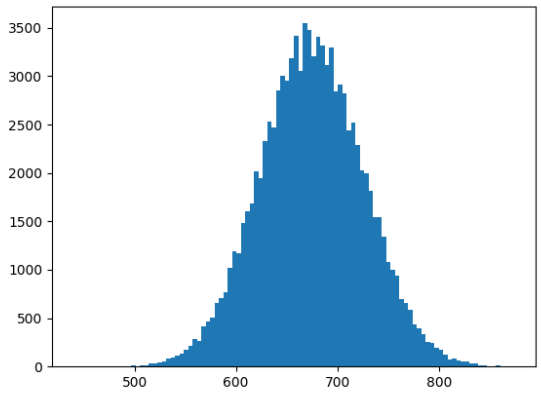



Figure 22.4: A histogram of bootstrap samples of the mean difference between male and female Adelie penguins.

## 22.3 Exercises

1. Write a function that simulates a single round of the Australian gambling game of two-up. In this game, the "spinner" essentially throws two coins in the air. If both coins come up heads, then the spinner wins. If both coins come up tails, the spinner loses. If they come up odds (one head, one tail), then neither the spinner nor the banker wins. The function should return one of the strings 'heads', 'tails', or 'odds' for a single spin.

2. Write a program that makes use of the function written in the previous question that simulates a full round of two-up. In a full round, the spinner keeps spinning until they either throw heads and win, or throw tails and lose. Output of the program should look something like the following:

```
Spinner throws odds
Spinner throws odds
Spinner throws heads.
Spinner wins!
```

3. Extend the program from the previous question to include an extra casino rule giving the casino a long run advantage. This extra rule is that the house wins if the spinner throws tails, or if they throw 5 odds in a row.

4. Using the penguins data, test the hypothesis that Gentoo penguins are heavier on average than Adelie penguins.

5. Repeat the previous question, but this time for Adelies and Chinstraps.

6. Find the 95% confidence interval for the difference in median values between Adelie male and female penguins.

7. Adelie penguins nest on three islands (Torgersen, Biscoe, Dream), test if there is a difference in average weight of Adelies between each pair of islands.

8. **Extension exercise:** Write a program to simulate the U.S.A gambling game of craps. The basic rules are:

- The player throws two dice initially. If the total comes up 7, the player wins. If the total comes up 2, 3 or 12, the player loses.
- If the player has not yet won or lost, then they get to keep playing. In this case, a winning throw is any throw that produces the same sum as the first throw. A losing throw is if the player throws a 7. The player keeps throwing until one of those conditions become true. For example, the play might go like this:

```
Player throws a 3 and a 2: 5
Player throws a 1 and a 1: 2
Player throws a 4 and a 5: 9
Player throws a 4 and a 1: 5. Player wins!
```

9. **Extension exercise:** Write a program for playing blackjack.

# GNU Free Documentation License

Version 1.2, November 2002



51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA



### Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the

Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name

of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Index