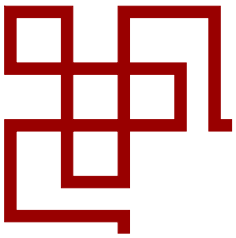
SMM692
NOTES

2022

Introduction to Programming in Python

Dr. Simone Santoni
Bayes Business School
106 Bunhill Row, London, EC1Y 8TZ



Version
Current version is 1.0 — last edited on August 21^{st} 2022

Contents

List of Figures

List of Tables

Preface

These notes were prepared for the Summer Module 'SMM-692 — Introduction to Programming in Python.' The materials I present condense more than ten years of experience I developed using Python for industrial applications and research. At the same time, the materials incorporate the feedback from four different cohorts of Bayes MSc students who took SMM692 and other modules for which Python is a prerequisite, such as 'SMM638 — Network Analytics,' 'SMM635 — Data Visualization,' and 'SMM694 — Natural Language Processing.'

I prepared these notes with two design principles in mind. First, the notes should have mimicked a conversation between a person interested in learning Python and a knowledge recipient (a robot or a human being, it does not matter!). The Cornell Notes Template helped me emphasize the notes' conversational structure: the left-hand side of the page is reserved for questions, reported as blue boxes; answers are displayed on the right-hand side of the page.

Second, I designed the notes to be as practical as possible. For this reason, the knowledge recipient's answer is supported by a Python code snippet. In total, there are 76 snippets the learner may want to scrutinize, reason with, run, and modify.

What to say? Have fun!

—Simone Santoni

Chapter 1

Organization of the Notes and SMM692

In this chapter, the reader is supposed to appreciate:

- The justification for an introductory module on Python for Business Analytics students
- The scope of the teaching materials included in these notes
- The learning and teaching activities that form 'SMM692 — Introduction to Programming in Python

---

1.1 Justification for an Introductory Module on Python

Why should I learn Python?

Python is a general-purpose, high-level programming language. So, why should business professionals learn it? Mainly, there are two reasons:

- First, Python is the center of one of the richest ecosystems of modules[1] for technical and scientific computation, which plays a key role in the quantitative analysis of organizations and markets.
- Second, for a large number of developers and users, 'Python' and 'data science' are largely overlapping skills (as Figure 1.1 shows, the popularity surge of Python is strictly related to the emergence and development of the data science field, YR 2014 →)

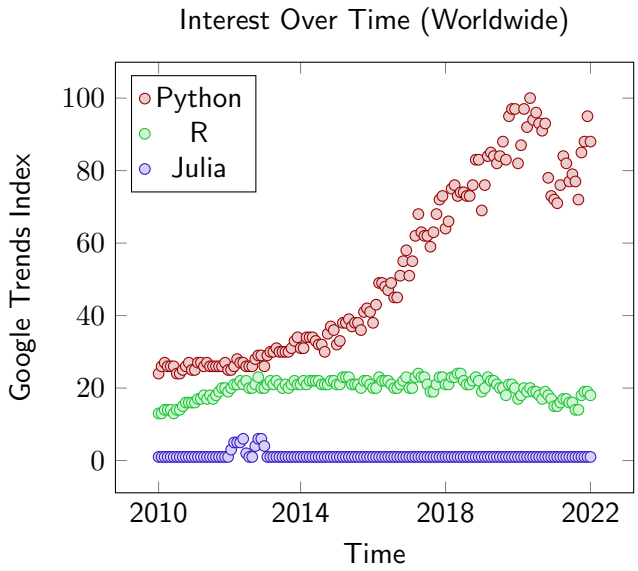


Figure 1.1: Interest towards data-science related programming languages over time
Notes. — Source is Google Trends

Why do I need to take a pre-course module on Python?

There are at least three arguments for taking a module such as SMM692:

- Modern quantitative curricula increasingly depart from a spreadsheet-based approach to teaching and learning
- For example, the Bayes Business Analytics MSc builds heavily on the Python programming language
  - There are *four core modules* based on Python: Data Visualization, Deep Learning, Network Analytics, Value Creation in Digital Settings
  - Plus *two elective modules*: Applied Machine Learning and Applied Natural Language Processing
- You may want to get up and running with Python in advance, that is, before the start of Term I

1.2 Scope of the Notes

What is the scope of SMM692?

The scope of SMM692 has been tailored to the learning needs of the Bayes MSc students. Specifically, these notes were created based on the feedback comments provided by four cohorts of students. Hence, I warmly invite you to focus on these notes instead of learning Python using the many books and resources available on the web — otherwise, you may miss core notions and tools or learn irrelevant ones!

So, do these notes cover the entire spectrum of the Python language?

NO! Covering the entire spectrum of the language would take substantial time — Python has extensive language references!! Also, some aspects of the Python language reference have limited added value for business professionals closer to Python users than developers.

Got it. Which parts of the Python language will I learn then?

The attention revolves around the following subjects:

- Variables
- Object
- Some built-in functions and methods for strings, lists, dictionaries, and sets

Learning the basics of the Python language will put you in the position to carry out technical and scientific computation tasks — namely, the essence of data science X BA

Which technical and scientific modules will I learn in these notes?

Two foundational modules support statistical analysis, Viz, ML, and DL: NumPy and Pandas. These two modules are the focus of SMM692's second part.

Can you summarize the subjects covered in these notes?

Of course. These notes cover the following topics:

- Getting started with Python (Chapter 2)
- Python objects (Chapter 3)
- Technical and scientific computation with NumPy (Chapter 4)
- Data management with Pandas (Chapter 5)

1.3 Learning and Teaching Activities

What is the philosophy behind these notes?

Mainly, there are two pillars:

- LESS IS MORE — that is, you may want to focus on a few core Python notions at a time and practice them
- LEARN BY DOING — that is, the best way to learn Python is by addressing concrete problems

I am a Programming Newbie: How Do I Learn Python?

Here is an approach that proved effective according to the pedagogical literature on learning programming languages:[2]

- Step 1: Focus on *few core notions* at a time (e.g., 'string methods')
- Step 2: Learn the selected core notions 'pen & paper' (stay away from the computer!)
- Step 3: Open a Python session and practice the few core notions
- Step 4: Critically self-assess your learning (if your code is not working, Python will force you to understand what's wrong!)

How do I assess my learning outcome?

You may want to assess your learning frequently, at the end of each chapter or, possibly, every two or three sections of the chapter. As Figure 1.2 suggests, the key idea is to adopt a sequential approach. Subject-by-subject, you may want to:

- Open a learning circle by watching the hook video and study the notes closely
- Test your understanding by taking a quiz
- If your learning is satisfactory, try to solve some practical coding problems (i.e., problem sets)

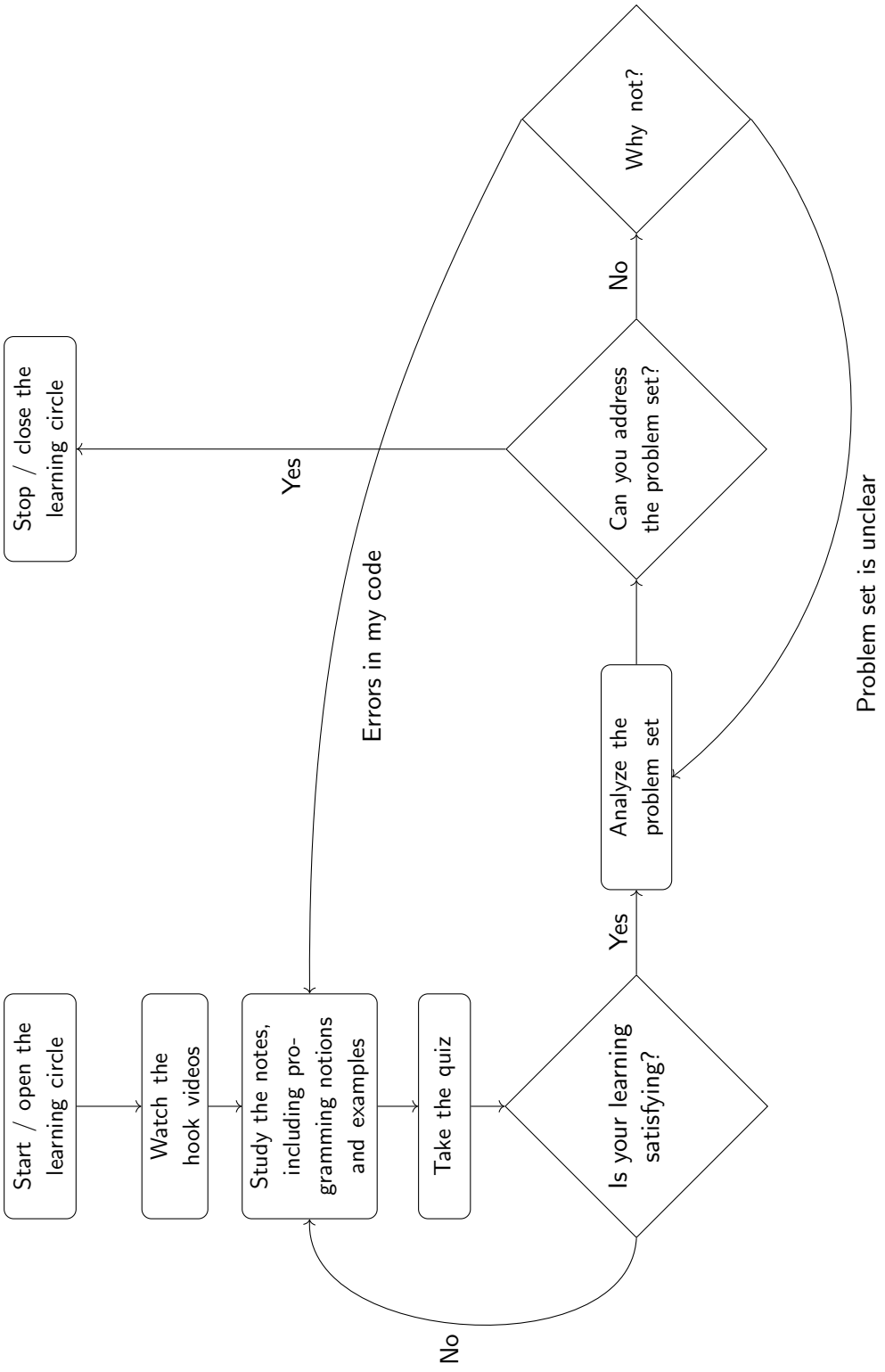


Figure 1.2: The suggested workflow to learn to code

Notes

[1]Throughout the various chapters, I will use the terms module, library, and package interchangeably.
[2]See for example i) Mayer, Richard E. 1981. "The Psychology of how Novices Learn Computer Programming." *ACM Computing Surveys* (CSUR) 13 (1): 121–141, ii) Robins, Anthony, Janet Rountree, and Nathan Rountree. 2003. "Learning and Teaching Programming: A Review and Discussion." *Computer Science Education* 13 (2): 137–172, and iii) Soloway, Elliot, and James C Spohrer. 2013. *Studying the Novice Programmer*. PsychologyPress.

Chapter 2

Getting Started with Python

At the end of the chapter, you will be able to:

- Get Python work in your machine
- Run Python code from a script file and interactively
- Configure a Python environment

2.1 Installing Python

How can I install Python?

There are two options:

- Using the official installer
- Using the Anaconda Distribution of Python (preferred option)

What are the distinctive features of Anaconda?

- Anaconda is 'battery-included' — it comes with a humongous number of modules for data science
  - If you use the official Python installation, you must install the modules you need on your own!
- Anaconda is a bundle of various pieces of software:
  - `conda` is the Swiss army knife to manage Python modules and environments
  - Anaconda Navigator is the graphical interface from within to access Python IDEs and related desktop/web applications

What are the steps to install Anaconda?

Here are the steps:

1. Downaload the installer for your operating system (unless you have a very old machine running Windows, go for the 64-Bit version)
2. Run the installer
   - For Linux: navigate to the folder where you have downloaded the installer as per step 1, open a shell session, then run `$ bash ./Anaconda3-XXXX.XX-Linux-x86-64.sh`
   - For Windows and Mac OS: just run the graphical installer downloaded in step 1
3. Accept the terms proposed by the Anaconda people to use their software, comprising Python, the `conda` package manager, and a bundle of modules for data science
4. That's it!
   - For Linux users: if you accepted the default installation options, an environmental variable is created either in your `.bashrc` or `.zshrc`. That means you can access the various pieces of software included in the Anaconda installation (e.g., Anaconda Navigator) from a shell session
   - For Windows and Mac OS users: the various pieces of software included in the Anaconda installation are available from the menu of your system

What are the pieces of software included in Anaconda?

There are plenty of applications included in the Anaconda installation. These applications can be accessed from within Anaconda Navigator (see Figure 2.1), available in the launcher of your operating system.

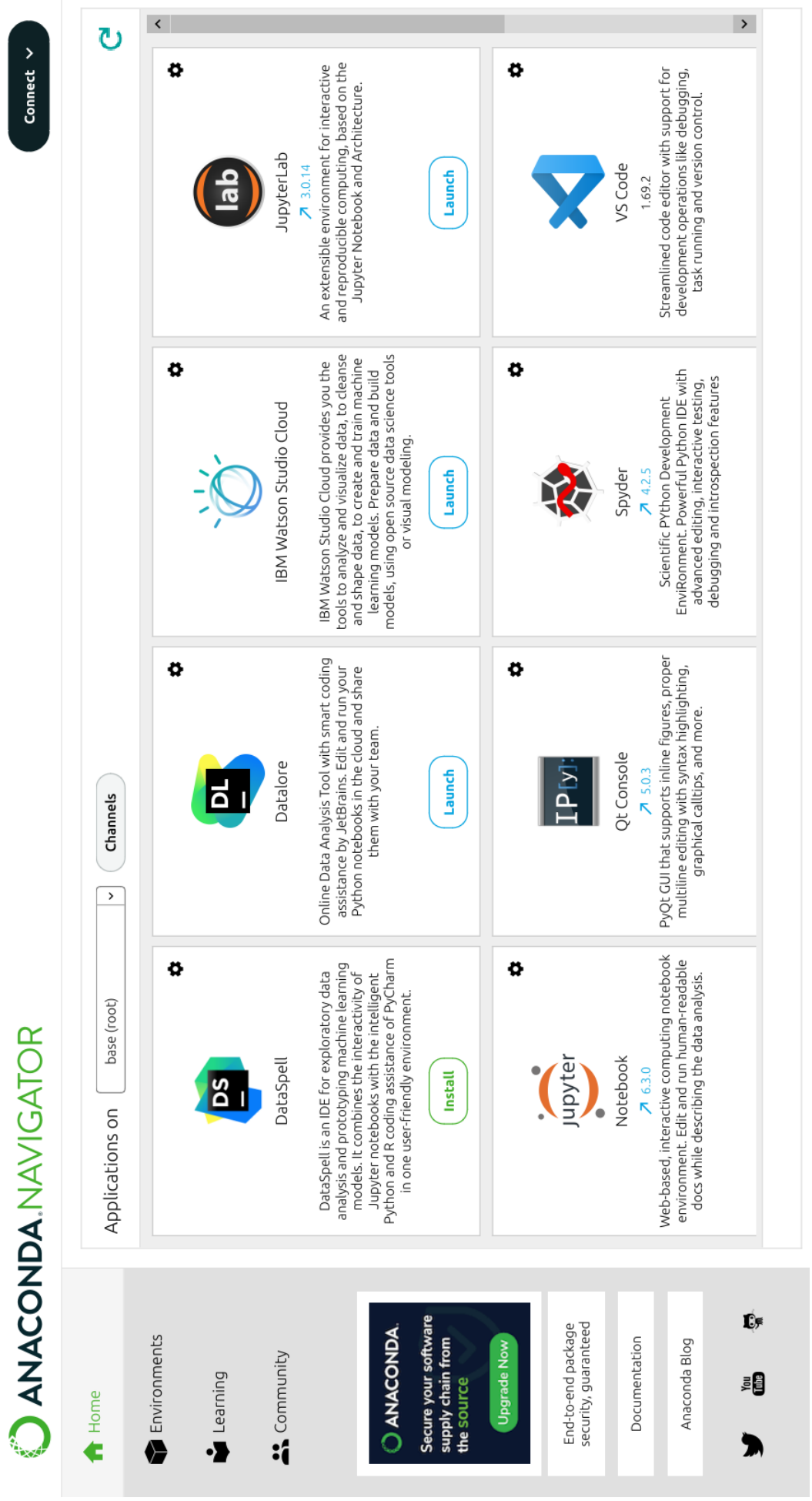


Figure 2.1: A screenshot showing a sample of applications accessible from within Anaconda Navigator

2.2 How Python Runs Programs

Is Python a programming language or an interpreter?

Short answer: BOTH. Typically, we refer to Python as a programming language. However, Python is also a software package called an interpreter, a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

How does an interpreter work?

- When you instruct Python to run your script (e.g., 'script.py'), there are a few steps the interpreter carries out before your code starts crunching away (see Figure 2.2)
- First, the code is compiled into something called 'byte code.'
  - This step happens behind the scenes (there is nothing to do for the programmer!)
  - A file called 'script. pyc', automatically generated, contains the translation of your code into lower-level code instructions
- Then, the compiled code is routed to something called a 'Python Virtual Machine' (PVM)
  - This step is hidden from the programmer like the previous one
  - Mainly, it iterates through your byte code instructions, one by one, to carry out their operations

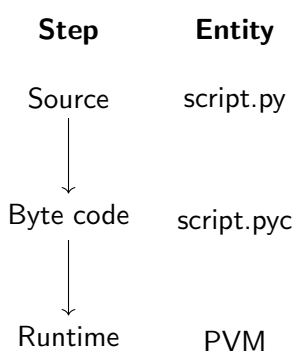


Figure 2.2: A stylized representation of Python's execution model

2.3 How We Run Python Programs

What are the alternative ways to run some Python code?

There are two alternatives:

- The non-interactive way consists of preparing a Python script and running it from the command line
- The interactive way consists of preparing and running one or a few Python statements from within a Python shell

How do I run a Python script in a non-interactive way?

One has to go through a two-step process:

- First, the script has to be prepared using a text editor (being basic or sophisticated, it is up to you)
- Second, the script has to be executed from the shell

Snippet 1.1 shows a minimal Python script. Such a script can be created using the built-in text editor of your operating system or an advanced text editor such as Atom, Emacs, Vim/Neovim, Sublime Text, and Visual Studio Code.

Once the script is saved to a file with extension `.py`, you can execute it from the command line by typing the following command in the shell session of our choice:

```
$ python filename.py
```

Let us unpack that command. The `$` symbol denotes the statement has to be run in a shell session; the `python` command tells the shell to use the Python interpreter to evaluate the script; `filename.py` is the script's file name. Figure 2.3 shows how to run the Python script containing the code displayed in Snippet 1.1 (named `simple_script.py`). [3] It is self-evident the outcome of the script is displayed in the active shell session (we will see the `print` function in action many times over the next few chapters).

Snippet 1.1 — a minimal Python script

```
1 # Print a string object
2 print("Bazinga")
3
4 # Print the result of an algebraic operation
5 print(2 + 4)
```

Figure 2.3: A screenshot showing the execution of the minimal script included in Snippet 1.1

How do I interactively run a Python script?

Mainly, there are three ways to do that:

- Running a Python shell in the terminal
- Running an IPython shell in the terminal
- Interacting with a Python or IPython shell through an Integrated Development Environment (IDE)

How do I run Python code interactively using a Python or IPython shell?

The first step is opening the terminal emulator of your choice (e.g., Windows Terminal, Cmder, Iterm, Terminator, Kitty). Then, you run the command `$ python` or `$ ipython` to start a Python and IPython shell respectively (see Figures 2.4 and 2.5).

Figure 2.4: A screenshot of an interactive Python shell

Figure 2.5: A screenshot of an interactive IPython shell

What are the most popular Python IDEs?

There are plenty of Python IDEs in the market, including:

- Colab (online)
- IDLE
- Datalore (online)
- Jupyter/Jupyterlab
- PyCharm
- Qt Console
- Spyder
- Thonny
- Wing

Shall I start using Jupyter?

Based on my experience, Jupyter sustains novices' programming learning in at least two ways. First, the graphical interface of a Jupyter notebook (see Figure 2.6 allows learners to embrace a trial and error approach to coding by dissecting a script into manageable snippets — or even single lines — that can be better examined, understood, and tested. Second, the text boxes and rich media content typically included in a notebook provide learners with explanations to understand a script's logical structure and background.

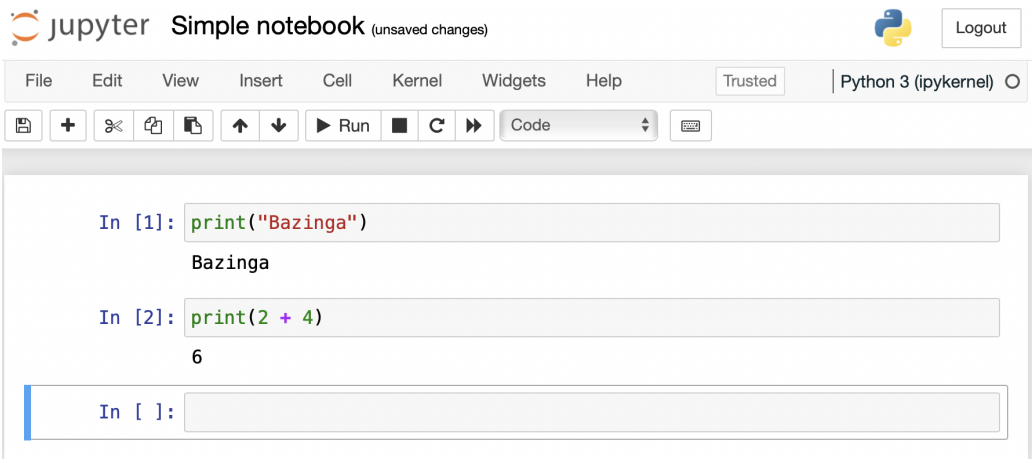


Figure 2.6: A screenshot of a Jupyter notebook session

Can I turn an advanced text editor into a Python IDE?

Of course. By installing a couple of plugins, the following (advanced) text editors can turn into Python IDEs

- Emacs
- Vm/Neovim
- Visual Studio Code (VSCode)

Figures 2.7 and 2.8 show how to run an IPython and Jupyter session in VSCode respectively.[4]

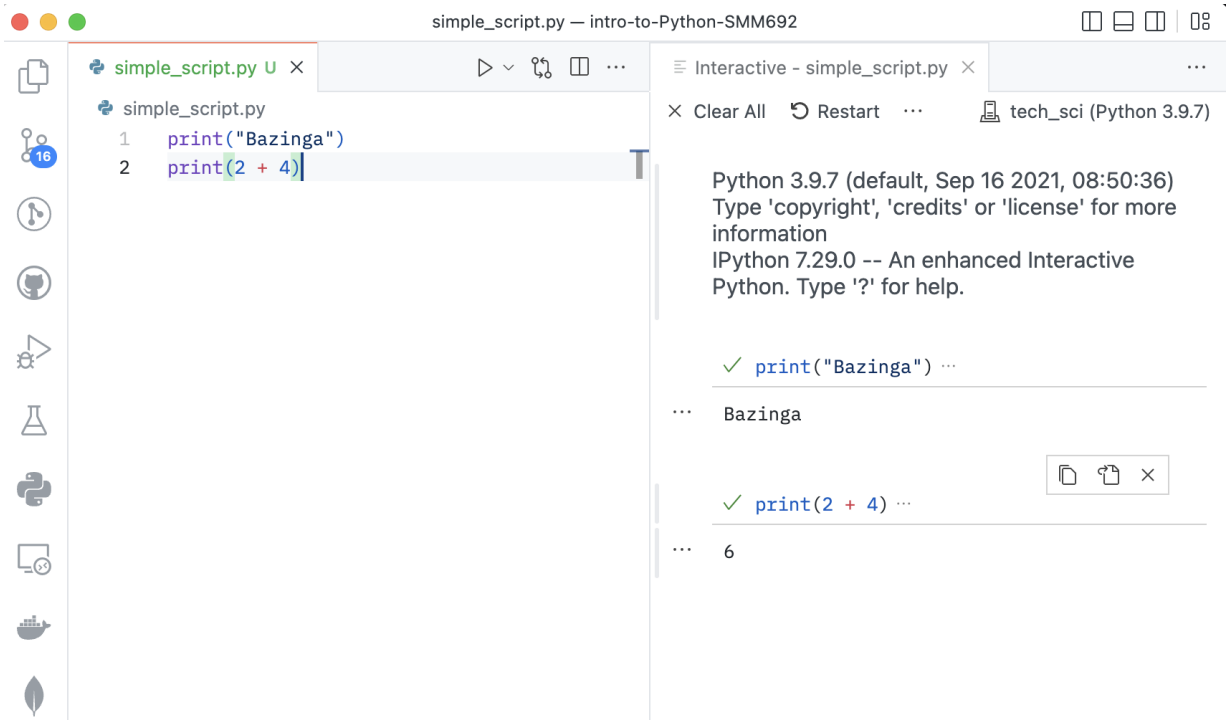


Figure 2.7: A screenshot of an interactive IPython shell in VSCode

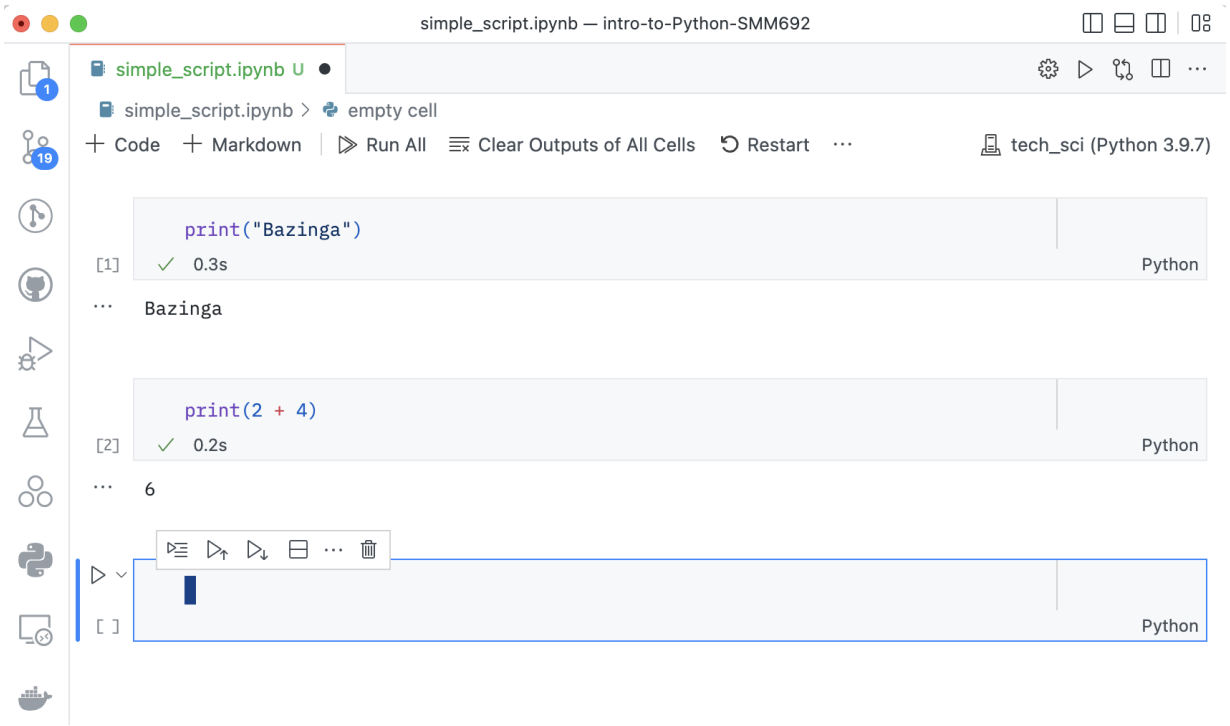


Figure 2.8: A screenshot of a Jupyter notebook session in VSCode

2.4 Managing Python Environments

What is a Python environment?

A Python environment is a self-contained directory that contains a Python interpreter, a set of Python packages, and their dependencies.

Why should I use a Python environment?

Mainly, there are two reasons:

1. To protect the system-wise installation of Python
2. To create collections of Python modules to deploy in specific projects/classes of projects (e.g., Machine Learning projects)

Why should I protect my system Python?

The large majority of operating systems come with Python installed by default. Such an installation is called 'system Python' and is responsible for many essential operating systems processes.

Every time we install a new Python module A, the package manager (e.g., conda) checks the libraries on which A depends. If a dependency $D(A)$ is not installed, the package manager will install it for us. The larger the number of libraries:

- The longer it takes to install a library (checking the web of dependencies takes substantial time!)
- The more likely inconsistencies will arise within the web of dependencies; that is
  - The less likely we will be able to install the modules we need
  - The more likely modules previously installed will stop work

Ultimately, if you care about not breaking some essential operating system features, you should avoid using 'system Python' for projects requiring Python.

Why Python environments work well for specific projects?

Typically, we create a Python environment to carry out a specific project or a class of projects (e.g., Machine Learning projects). The advantage is twofold:

- Reliability → the web of dependencies is relatively simple insofar as we install only a few modules that are required by the project or project class. Likely as not, we will not come across installation issues!
- Reproducibility → the environment explains the modules necessary to carry out a project. Hence, a user who wants to reproduce our project's results knows which modules to install

How do I create a Python environment?

The procedure to create a Python environment depends on the package manager you use (hence, the Python installation we have). Here, I focus on the case where conda is the adopted package manager.[5]

How do I use `conda` from the command line to create and populate a Python environment?

To create a Python environment using `conda`, we can run the following command in the shell:

```
$ conda create --name myenv python=3.X
```

where `conda` is the name of the package manager, `create` is the operation to run,[6] `myenv` is the name of the environment and `python=3.X` is the version of Python to use (e.g., 3.10).

To use the newly created environment, one has to activate it by running the following command in the shell:

```
$ conda activate myenv
```

At this point, one can install packages in the activated environment using the following command:

```
$ conda install package_name
```

Figures 2.9 and 2.10 show a concrete example wherein a new Python environment is created and populated using `conda` from the command line.

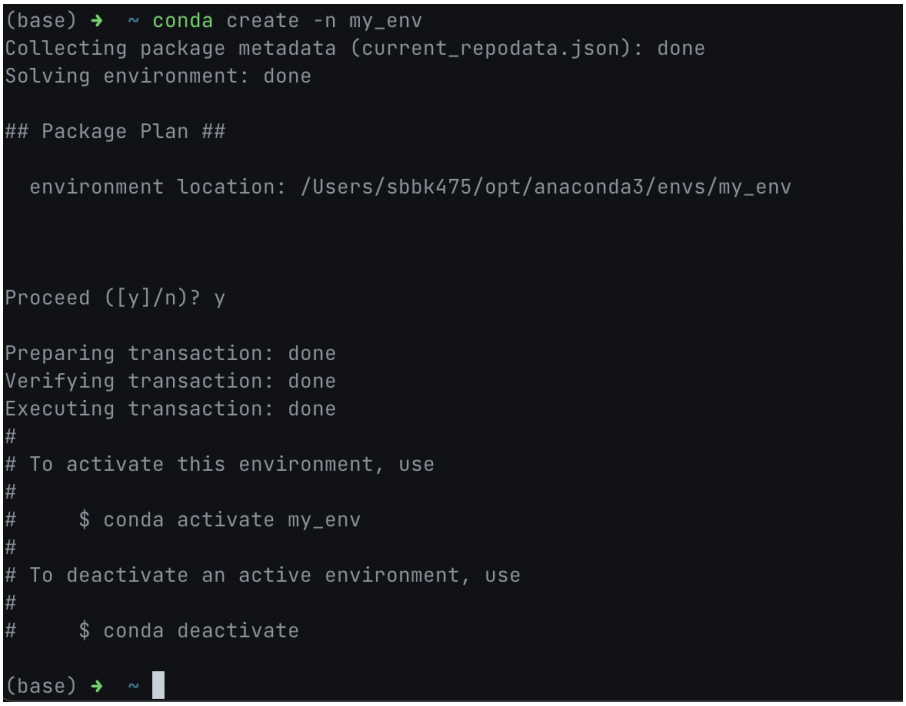


Figure 2.9: A screenshot showing how to create a Python environment using `conda` from the command line

Figure 2.10: A screenshot showing how to populate a Python environment with libraries using `conda` from the command line

How do I use Anaconda Navigator to create and populate a Python environment?

It is also possible to create a Python environment from within the Anaconda Navigator (Figures 2.11 and 2.12 provide a pictorial illustration of how to create a new environment and populate it with modules).

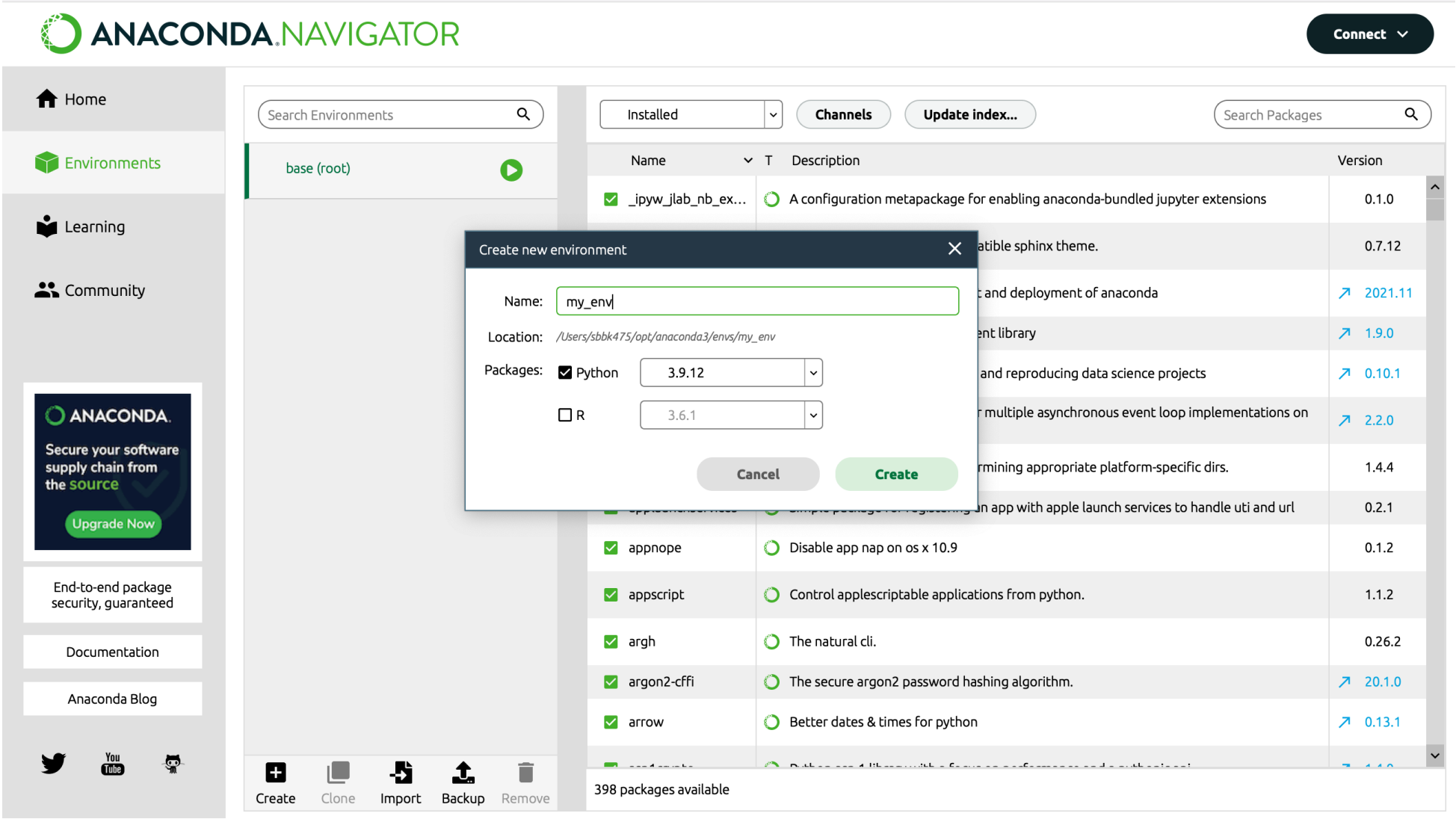


Figure 2.11: A screenshot showing how to create a new environment from within the 'Environments' section of Anaconda Navigator

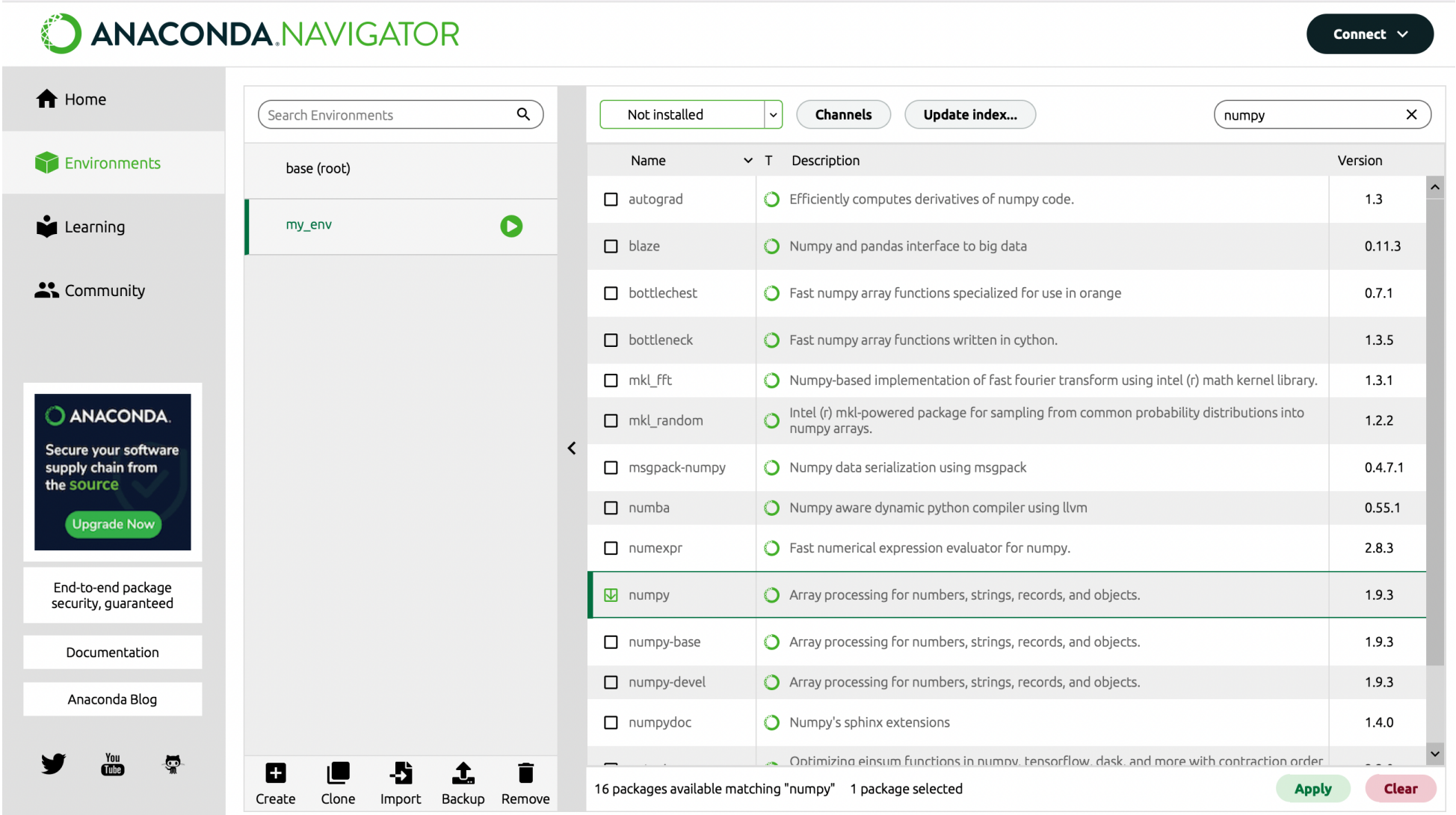


Figure 2.12: A screenshot showing how to add Python modules to a (newly created) environment from within the 'Environments' section of Anaconda Navigator

Notes

[3]The example shown in the screenshot assumes the Python script `sample_script.py` is saved in the current working directory. If the script is saved in a different directory, you have to specify the full path to the script in the command line.
[4]Please refer to the official documentation on how to use Python in VSCode.
[5]`pip` users are warmly encouraged to read the Virtual Environments and Packages section of the official Python Tutorial.
[6]The Conda Command reference illustrates the various commands Conda provides for managing packages and environments.

Chapter 3

Python Objects

At the end of the chapter, you will be able to evaluate the various types of Python objects regarding:

- Key features
- Use cases/roles
- Available methods

What is a Python object?

In essence, Python objects are pieces of data. Mark Lutz, the author of the popular book Learning Python[7], points out

> "*... in Python, we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations*".

What are the main families of Python objects?

In Python, there are two families of objects: built-in objects provided by the Python language itself and ad-hoc objects — called classes — we can create to accomplish specific goals.

Why do built-in Python objects matter?

Typically, we do not need to create ad-hoc objects. Python provides us with diverse built-in objects that make our job easier:

- Built-in objects make coding efficient and easy. For example, using the string object, we can represent and manipulate a piece of text — e.g., a newspaper article — without loading any module
- Built-in objects are flexible. For example, we can deploy built-in objects to create a class
- Built-in objects have been created and refined over time by a large community of expert developers. Hence, they are often more efficient than ad-hoc objects (unless the creator of the ad-hoc object knows her business!)

What are the core built-in Python objects?

Table 3.1 illustrates the types of built-in Python objects. For example, Numbers and strings objects are used to represent numeric and textual data respectively. Lists and dictionaries are — likely as not — the two most popular data structures in Python. Lists are ordered collections of other objects (any type!!). Dictionaries are pairs of keys (e.g., a product identifier) and objects (e.g., the product's price). No worries: we will go through each built-in type in the following sections of this document. Caveat: in the interest of logical coherence, the various built-in types will not be presented in the order adopted in Table 3.1.

TABLE 3.1
Built-In Objects in Python

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation types	Compiled code, stack tracebacks

3.1 Number Type Fundamentals

What are the types of 'number' objects?

Snippet 4.1, "Doing stuff with numbers," highlights the two most popular 'number' instances in Python: integers and floating-point numbers. Integers are whole numbers such as 0, 4, or -12. Floating-point numbers represent real numbers such as 0.5, 3.1415, or -1.6e-19. However, floating points in Python do not have — in general — the same value as the real number they represent.[8] It is worth noticing that any single number with a period '.' is considered a floating point in Python. Also, Snippet 4.1 shows that the multiplication of an integer by a floating point yields a floating point. That happens because Python first converts operands to the type of the most complicated operand.

Snippet 4.1 — doing 'stuff' with numbers

```
# integer addition
>>> 1 + 1
2

# floating-point multiplication
>>> 10 * 0.5
5.0

# 3 to the power 100
>>> 3 ** 100
515377520732011331036461129765621272702107522001
```

Are there other number types besides integers and floating points?

Besides integers and floating points numbers, Python includes fixed-precision, rational numbers, Booleans, and sets instances — see Table 3.2.

TABLE 3.2
Number Type Objects in Python

Literal	Interpretation
1234, -24, 0, 99999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.X
0177, 0o177, 0x9ff, 0b101010	Octal, octal, hex, and binary literals in 2.X
3+4j, 3.0+4.0j, 3J	Complex number literals
set('spam'), {1, 2, 3, 4}	Sets: 2.X and 3.X construction forms
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

How do I carry out basic arithmetic operations in Python?

Numbers in Python support the usual mathematical operations:

- `+` → addition
- `-` → subtraction
- `*` → multiplication
- \ → floating point division
- `//` → integer division
- `%` → modulus (remainder)
- `**` → exponentiation

To use these operations, it is sufficient to launch a Python or IPython session without any modules loaded (see Snippet 4.1).

How do I carry out advanced mathematical operations?

Besides the mathematical operations shown above, there are many modules shipped with Python that carry out advanced/specific numerical analysis. For example, the `math` module provides access to the mathematical functions defined by the C standard.[9] Table 3.3 reports a sample of these functions. To use them `math`, we have to import the module as shown in Snippet 4.2. Another popular module shipped with Python is `random`, implementing pseudo-random number generators for various distributions (see the lower section of Example 2).

TABLE 3.3
A Sample of Functions Provided by the `math` Module

Function name	Expression
`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	lnx
`math.log(x, b)`	$log_b(x)$
`math.log10(x)`	$log_{10}(x)$
`math.sin(x)`	$sin(x)$
`math.cos(x)`	$cos(x)$
`math.tan(x)`	$tan(x)$
`math.asin(x)`	$arcsin(x)$
`math.acos(x)`	$arccos(x)$
`math.atan(x)`	$arctan(x)$
`math.sinh(x)`	$sinh(x)$
`math.cosh(x)`	$cosh(x)$
`math.tanh(x)`	$tanh(x)$
`math.asinh(x)`	$arsinh(x)$
`math.acosh(x)`	$arcosh(x)$
`math.atanh(x)`	$artanh(x)$
`math.hypot(x, y)`	The Euclidean norm, $\sqrt{x^2 + y^2}$
`math.factorial(x)`	$x!$
`math.erf(x)`	The error function at x
`math.gamma(x)`	The gamma function at x, $\omega(x)$
`math.degrees(x)`	Converts x from radians to degrees
`math.radians(x)`	Converts x from degrees to radians

Snippet 4.2 — advanced mathematical operations with the math module

```
1  # import the math module
2  >>> import math
3
4  # base-y log of x
5  >>> math.log(12, 8)
6  1.1949875002403856
7
8  # base-10 log of x
9  >>> math.log10(12)
10 1.0791812460476249
11
12 # import the random module
13 >>> import random
14
15 # a draw from a normal distribution with mean = 0 and standard deviation = 1
16 >>> random.normalvariate(0, 1)
17 -0.136017752991189
18
19 # trigonometric functions
20 >>> math.cos(0)
21 1.0
22
23 >>> math.sin(0)
24 0.0
25
26 >>> math.tan(0)
27 0.0
28
29 # an expression containing a factorial product
30 >>> math.factorial(4) - 4 * 3 * 2 * 1
31 0
```

What is the precedence order among Python operators?

As shown in Snippet 4.2, line 30, Python expressions can string together multiple operators. So, how does Python know which operation to perform first? The answer to this question lies in operator precedence. When you write an expression with more than one operator, Python groups its parts according to what is called precedence rules,[10] and this grouping determines the order in which the expression's parts are computed. Table 3.4 reports the precedence hierarchy concerning the most common operators. Note that operators lower in the table have higher precedence. Parentheses can be used to create sub-expressions that override operator precedence rules.

TABLE 3.4
Operator Precedence Hierarchy
(Ascending Order)

Operator	Description
`x + y`	Addition, concatenation
`x - y`	Subtraction, set difference
`x * y`	Multiplication, repetition
`x % y`	Remainder, format;
`x / y`, `x // y`	Division: true and floor
`-x`, `+x`	Negation, identity
`~x`	Bitwise NOT (inversion)
`x ** y`	Power (exponentiation)

How do I carry out technical and scientific computation with Python?

Python is at the center of a rich ecosystem of modules for technical and scientific computation. In the following chapter, the attention will revolve around one of the most prominent modules, namely, NumPy. In a nutshell, NumPy offers the infrastructure for efficiently manipulating data structures. SciPy builds on NumPy to implement many algorithms across the fields of statistics, linear algebra, optimization, calculus, signal processing, image processing, and others. Another core module in the technical and scientific domain is efficiently manipulatingSimPy

, a library for symbolic mathematics. Note that none of these three modules are shipped with Python and should be installed with the package manager of your choice (e.g., `conda`).

What is a Python variable?

Variables are simply names — created by you or Python — that are used to keep track of information in your program. In Python:

- Variables are created when they are first assigned values
- Variables are replaced with their values when used in expressions
- Variables must be assigned before they can be used in expressions
- Variables refer to objects and are never declared ahead of time

As Snippet 4.3 shows, the assignment of `x = 2` causes the variable `x` to come into existence 'automatically.' From that point, we can use the variables in the context of expressions such as the ones displayed in lines 8, 12, 16, and 20 or create new variables like in line 24.

Snippet 4.3 — expressions involving arithmetic operations

```
1
2  # let us assign the variables 'x' and 'y' to two number objects
3  >>> x = 2
4
5  >>> y = 4.0
6
7  # subtracting an integer from variable 'x'
8  >>> x - 1
9  1
10
11 # dividing the variable 'y' by an integer
12 >>> y / 73
13 0.0547945205479452
14
15 # integer-dividing the variable 'y' by an integer
16 >>> y // 73
17 0.0
18
19 # getting a linear combination of 'x' and 'y'
20 >>> 3 * x - 5 * y
21 -14.0
22
23 # assigning the variable 'z' to the linear combination of 'x' and 'y'
24 >>> z = 3 * x - 5 * y
```

How do I display number objects in a readable way?

Snippet 4.3 includes some expressions whose result is not passed to a new variable (e.g., lines 8, 12, 16, 20). In those cases, the IPython session displays the expression's outcome as is (e.g., 0.0547945205479452). However, a number with more than three or four decimals may not suit the table or report we must prepare. Python has powerful string formatting capabilities to display number objects in a readable and nice manner. Table 3.5 illustrates various number formatting options with concrete cases. Format strings contain 'replacement fields' surrounded by curly braces {}. Anything not contained in braces is considered literal text, copied unchanged to the output. Snippet 4.4 presents a fully-fledged number formatting case. First, we assign the variable `a` to a floating-point number (line 2). Then, we pass the formatting option {`:.2f`} over the variable `a` using the Python built-in function `format`.

TABLE 3.5
Number Formatting Options in Python

Number	Format	Output	Description
3.1415926	{:.2f}	3.14	Format float 2 decimal places
3.1415926	{:+.2f}	+3.14	Format float 2 decimal places with sign
-1	{:+.2f}	-1.00	Format float 2 decimal places with sign
2.71828	{:.0f}	3	Format float with no decimal places
5	{:0>2d}	05	Pad number with zeros (left padding, width 2)
5	{:x<4d}	5xxx	Pad number with x's (right padding, width 4)
10	{:x<4d}	10xx	Pad number with x's (right padding, width 4)
1000000	{:,}	1,000,000	Number format with comma separator
0.25	{:.2%}	25.00%	Format percentage
1000000000	{:.2e}	1.00e+09	Exponent notation
13	{:10d}	13	Right aligned (default, width 10)
13	{:<10d}	13	Left aligned (width 10)
13	{:^10d}	13	Center aligned (width 10)

Snippet 4.4 — number formatting examples

```
1  # assign the variable 'a' to a floating-point number
2  >>> a = 0.67544908755
3
4  # displaying 'a' with the first two decimals only
5  >>> "{:.2f}".format(a)
6  "0.68"
7
8  # displaying 'a' with the first three decimals only
9  >>> "{:.3f}".format(a)
10 "0.675"
```

How do I compare number objects?

Comparisons are used frequently to create control flows, a topic we will discuss later in this chapter. Normal comparisons in Python regard two number objects and return a Boolean result. Chained comparisons concern three or more objects and, like normal comparisons, yield a Boolean result. Snippet 4.5 provides a sample of normal comparisons (between lines 1 and 15) and chained comparisons (between lines 21 and 30). As evident in the example, comparisons can regard both numbers and variables assigned to numbers. Chained comparisons can take the form of a range test (see line 21), a joined, 'AND' test of the truth of multiple expressions (see line 25), or a disjoined, 'OR' test of the truth of multiple expressions (see line 29).

Snippet 4.5 — comparing numeric objects

```
1  # less than
2  >>> 3 < 2
3  False
4
5  # greater than or equal
6  >>> 1 <= 2
7  True
8
9  # equal
10 >>> 2 == 2
11 True
12
13 # not equal
14 >>> 4 != 4
15 False
16
17 # range test
18 >>> x = 3
19 >>> y = 5
20 >>> z = 4
21 >>> x < y < z
22 False
23
24 # joined test
25 >>> x < y and y > z
26 True
27
28 # disjoined test
29 >>> x < y or y < z
30 True
```

3.2 String Type Fundamentals

What is a string?

A Python string is a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative positions. Strictly speaking, strings are *immutable sequences* of one-character strings; other, more general sequence types include lists and tuples, covered later.

How do I use strings?

Strings are used to record words, contents of text files loaded into memory, Internet addresses, Python source code, and so on. Strings can also hold the raw bytes used for media files and network transfers and the encoded and decoded forms of non-ASCII Unicode text used in internationalized programs.

Is `abc` a Python string?

Nope. Python strings are enclosed in single quotes ('...') *or* double quotes ("...") with the same result. Hence, "`abc`" can be Python string, while `abc` cannot. `abc` can be a variable name, though.

How do I manipulate string objects?

The fact that strings are immutable sequences affects how we manipulate textual data in Python. In Snippet 4.6, we fetch the individual elements of `S`, a variable assigned to "`Python 3.X`." As per the built-in function `len`, `S` contains six unitary strings. That means that each element in `S` is associated with a position in the numerical progression $\{0, 1, 2, 3, 4, 5\}$.

Now, you may be surprised that the list's first element is 0 instead of 1. The reason is that Python is a zero-based indexed programming language: the first element of a series has an index of 0, while the last part has an index `len(obj) - 1`.

Fetching the individual elements of a string, such as `S`, requires passing the desired index between brackets, as shown in line 9 (where we get the first unitary string, namely, "`P`".), line 13 (where we get the last unitary string, namely, "`X`"), and line 21 (where we get the unitary string with index 3, i.e., the fourth unitary string appearing in `S`, "`h`"). Note that line 17 is an alternative indexing strategy to the one presented in line 13: it is possible to retrieve the last unitary string by counting 'backward'; that is, getting the first element starting from the right-hand side of the string, which equates to index `-1`.

In lines 26 and 31, we exploit the indices of `S` to retrieve multiple unitary strings in a row. What we pass among brackets is not a single index. Instead, we specify a range of indices `i:j`. It is worth noticing that, in Python, the element associated with the lower bound index is returned. In contrast, the element associated with the upper bound index `j` is not. In line 26, we fetch the unitary strings between index `2` — equating to third unitary string of `S` — and index `5` excluded — namely, the fifth unitary string of `S`. In line 30, we adopt the 'backward' approach to retrieve the unitary string with index `-3` — the third string counting from the right-hand side of `S` — as well as any other unitary strings following index `-3`. To do that, we leave the upper bound index blank.

Snippet 4.6 — Python strings as sequences

```
# let us assign the string "Python 3.X" to the variable S
>>> S = "Python 3.X"

# check the length of S
>>> len(S)
6

# access the first unitary string in the sequence behind S
>>> S[0]
"P"

# access the last unitary string in the sequence behind S
>>> S[len(S)-1]
"X"

# or, equivalently
>>> S[-1]
"X"

# access the i-th, e.g., 3rd, unitary string in the sequence behind S
>>> S[3]
"h"

# access the unitary strings between the i-th and j-th positions in the
# sequence behind S
>>> S[2:5]
"tho"

# access the unitary strings following the i-th position in the sequence
# behind S
>>> S[-3:]
"3.X"
```

What are the most common string literals and operators?

Snippet 4.6 deals with string indexing and slicing, two of the many operations we can carry out on strings. Table 3.6 reports a sample of common string literals and operators.

The first two lines of Table 3.6 remind us that single and double quotes are equivalent when assigning a variable to a string object. However, we must refrain from mixing and matching single and double quotes. In other words, a string object requires the leading and trailing quotes are of the same type (i.e., double-double or single-single).

In the interest of consistency, it is a good idea to make a policy choice, such as "in my Python code, I use double quotes only", and to stick with that throughout the various lines of the script. I prefer using double quotes because the single quote symbol is relatively popular in natural language (consider, for example, the Saxon genitive).

As shown in the third line of Table 3.6, the single quote is treated as a unitary string insofar as double quotes are used to delimit the string object. Should the string object be delimited by single quotes, we should tell Python not to treat the single quote symbol after `m` as a Python special character but as a unitary string. To do that, we use the escape symbol \ as shown in the fourth line of Table 3.6. Table A.1 provides several escaping examples.

Can you share some concrete examples of string manipulation tasks?

Snippet 4.7 presents a sample of 'common' miscellaneous string manipulation tasks. In lines 6 and 9, we check the length of the variables `S1` and `S2`. In line 13, we display five repetitions of `S1`. In line 17, we use the algebraic operator "+" to concatenate `S1` and `S2`. In line 21, we expand on the previous input by separating `S1` and `S2` by a white-space. In line 25, we carry out the same task as line 17 — however, we rely on the built-in `join` function to join `S1` and `S2` with whitespace. The argument taken by `join` is a Python `list`, the subject of paragraph 5.3. In line 29, `S1` and `S2` are joined with a custom string object, namely, `" Vs. "`. Finally, in line 33, we use the built-in `format` function (see also Snippet 4.4) to display a string object including `S1` and `S2`. For a comprehensive list of string methods, see Table A.2.

TABLE 3.6
Sample of String Literals and Operators

Literal/operation	Interpretation
`S = ""`	Empty string
`S = ‘’`	Single quotes, same as double quotes
`S = "spam’s"`	Single quote as a string
`S = ‘spam\’s’`	Escape symbol
`length(S)`	Length
`S[i]`	Index
`S[i:j]`	Slice
`S1 + S2`	Concatenate
`S * 3`	Repeat S n times (e.g., three times)
`"text".join(strlist)`	Join multiple strings on a character (e.g., “text”)
`"{}".format()`	String formatting expression
`S.strip()`	Remove white spaces
`S.replace("pa", "xx")`	Replacement
`S.split(",")`	Split on a character (e.g., “,”)
`S.lower()`	Case conversion — to lower case
`S.upper()`	Case conversion — to upper case
`S.find("text")`	Search substring (e.g., ”text”)
`S.isdigit()`	Test if the string is a digit
`S.endswith("spam")`	End test
`S.startswith("spam")`	Start test
`S = """...multiline..."""`	Triple-quoted block strings

Snippet 4.7 — miscellaneous string manipulation tasks

```
# let us assign S1 and S2 to two strings
>>> S1 = "Python 3.X"
>>> S2 = "Julia"

# check the length of S1 and S2
>>> len(S1)
10

>>> len(S2)
5

# display the S1 repeated five times
>>> S1 * 5
"Python 3.XPython 3.XPython 3.XPython 3.XPython 3.X"

# display the concatenation of S1 and S2
>>> S1 + S2
"Python 3.XJulia"

# display the concatenation of S1, whitespace, and S2
>>> S1 + " " + S2
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with a whitespace
>>> " ".join([S1, S2])
"Python 3.X Julia"

# display the outcome of joining S1 and S2 with an arbitrary string object
>>> " Vs. ".join([S1, S2])
"Python 3.X Vs. Julia"

# string formatting
>>> "Both {} and {} have outstanding ML modules".format(S1, S2)
"Both Python 3.X and Julia have outstanding ML modules"
```

Can you share some concrete examples of string editing tasks?

Snippet 4.8 illustrates some string editing tasks. In line 5, we use lstrip — a variation of the built-in function strip — that returns a copy of the string with leading characters removed. In line 9, we use the built-in replace to return a copy of the string with all occurrences of substring `old` (first argument taken by the function) replaced by `new` (second argument taken by the function). Finally, in line 17, we use the built-in function `lower` to return a copy of the string with all the cased characters converted to lowercase.

Snippet 4.8 — miscellaneous string editing tasks

```
# let us assign S to a string object
>>> S = "Both Python 3.X and Julia have outstanding ML modules"

# strip target leading characters
>>> S.lstrip("Both ")
"Python 3.X and Julia have outstanding ML modules"

# replace target characters
>>> S.replace("Python 3.X", "R")
"Both R and Julia have outstanding ML modules"

# split string on target characters
>>> S.split(" and ")
["Both Python 3.X", "Julia have outstanding ML modules"]

# make the string lower case
>>> S.lower()
"both python 3.x and julia have outstanding ml modules"
```

How do I test or search string attributes?

Snippet 4.9 presents a series of string test and search tasks. The built-in function `find` (see lines 5 and 9) returns the lowest index in the string where substring sub is found within the slice `S[start:end]` or -1 if substring is not found. The built-in function `isdigit` return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Finally, the built-in function endswith returns `True` if the string ends with the specified suffix, otherwise returns `False`.

Snippet 4.9 — miscellaneous string test and search tasks

```
# let us assign S to a string object
>>> S = "The first version of Python was released in 1991"

# search for "Python" in S
>>> S.find("Python")
21

# search for "Julia" in S
>>> S.find("Julia")
-1

# slice the string the get Python's release year information
>>> SS = S[-4:]

# display SS
>>> SS
"1991"

```

```
19 # test if all characters in SS are digits
20 >>> SS.isdigit()
21 True
22
23 # test if all characters in SS are digits
24 >>> SS.isdigit()
25 True
26
27 # test if all characters in SS are digits
28 >>> SS.isdigit()
29 True
30
31 # test if S ends with "1991" / SS
32 >>> S.endswith(SS)
33 True
```

Can I display 'complex' string objects?

We just came across the built-in function `print`. Such a function can print both number- and string-type objects. Sometimes, what we want to print fits into a single line. In other circumstances, we are interested in visualizing rich data which can span multiple lines. Snippet 4.9 how to print objects across multiple lines with the triple-quoted block string (see line). As evident from the Python code in lines 8-13, any line between triple quotes is considered part of the same string object.

Snippet 4.9 — multiline string printing

```
1  # single-line print
2  >>> print("Hello world!")
3  Hello world!
4
5  # multi-line print
6  >>> print(
7  ... """
8  ... =====================================================
9  ... COL A       | COL B      | ...        | COL K
10 ... -----------------------------------------------------
11 ... Sheldon     | Cooper     | ...        | bazinga.com
12 ... -----------------------------------------------------
13 ... NOTES: this table has fake data
14 ... """
15 ... )
16
17 =====================================================
18 COL A       | COL B      | ...        | COL K
19 -----------------------------------------------------
20 Sheldon     | Cooper     | ...        | bazinga.com
21 -----------------------------------------------------
22 NOTES: this table has fake data
```

3.3 List and Dictionaries

What is a list?

A Python `list` is an *ordered*, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, `[]`.

Why should I use lists?

Lists are just places to collect other objects so you can treat them as groups.

What type of objects can I include in a list?

Lists can contain any sort of object: numbers, strings, and even other lists. See Snippet 4.10.

Snippet 4.10 — sample lists with different items

```
1 # an empty list
2 >>> L = []
3
4 # a list with an integer, a float, and a string
5 >>> L = [2, -3.56, "XyZ"]
6
7 # a list with an integer and a list
8 >>> L = [4, ["abc", 8.98]]
```

How can I fetch a list's elements?

We can retrieve one or more list component objects via indexing. That is possible because list items are ordered by their position (similarly to strings). Since Python is a zero-based indexed programming language, to fetch the first item of a list, we have to call the index `0` (see Snippet 4.11, line 5). A list nested in another list can be fetched using multiple indices (line 13). The first index refers to the outer list, while any subsequent index refers to an inner list. In our case, we have two indices, one for each list; that is, `L` and its sub-list `["abc", 8.98]`.

Snippet 4.11 — list indexing and slicing

```
1  # the list
2  >>> L = [4, ["abc", 8.98]]
3
4  # get the first item of L
5  >>> L[0]
6  4
7
8   # get the second element of L
9  >>> L[1]
10 ["abc", 8.98]
11
12 # get the first item of L's second item
13 >>> L[1][0]
14 "abc"
```

Can I edit a list's elements?

Lists are mutable objects, which may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more. Snippet 4.12 illustrates some snippets to change a list's items. In the first part of the example, we change the items using indexing (line 5) and slicing (line 10). In the second half, we use Python's `del` statement to delete the items using indexing (line 15) and slicing (line 20).

Snippet 4.12 — changing and deleting list items in place

```
1  # the list
2  >>> L = ["Leonard", "Penny", "Sheldon"]
3
4  # change the second item of L via indexing
5  >>> L[1] = "Raj"
6  >>> print(L)
7  ["Leonard", "Raj", "Sheldon"]
8
9  # change multiple items of L via slicing
10 >>> L[0:2] = ["Amy", "Howard"]
11 >>> print(L)
12 ["Amy", "Howard", "Sheldon"]
13
14 # delete the first item of L via indexing and using the 'del' statement
15 >>> del L[0]
16 >>> print(L)
17 ["Howard", "Sheldon"]
18
19 # delete multiple items of L via slicing and using the 'del' statement
20 >>> del L[0:2]
21 >>> print(L)
22 []
```

What are the built-in methods to manipulate a list?

Python offers many methods to manipulate and test list objects. Table 3.7 reports some of the most popular methods and synopses. The first three methods — `.append()`, `.insert()`, and `.extend()` — expand an existing list. The fourth method, `.index` test for the presence of an item in an existing list. Note this method raises a `ValueError` if there is no such item.The remaining methods produce in-place changes in an existing list's items.

TABLE 3.7
Popular List Methods

Method	Synopsis
`L.append(X)`	Append an item to an existing list
`L.insert(i, X)`	Append an item to an existing list in position i
`L.extend([X0, X1, X2])`	Extend an existing list with the items from another list
`L.index(X)`	Get the index of the first instance of the argument in an existing list
`L.count(X)`	Get the cardinality of an item in an existing list
`L.sort()`	Sort the items in an existing list
`L.reverse()`	Reverse the order of the items in an existing list
`L.copy()`	Get a copy of an existing list
`L.pop(i)`	Remove the item at the given position in the list, and return it
`L.remove(X)`	Remove the first instance of an item in an existing list
`L.clear()`	Remove all items in an existing list

How can I expand an existing list?

Both the `.append()` and `.extend()` methods can be used to expand an existing list as per Snippet 4.13. However, they accomplish different goals and should not be confused: `.append()` adds a new item (of any type) to the end of the list (see line 6); `.extend()` extend the list by appending all the items from another iterable object (e.g., another list, see line 11).

Snippet 4.13 — methods for expanding an existing list

```
1  # create two lists
2  >>> L1 = ["Leonard", "Penny", "Sheldon"]
3  >>> L2 = ["Howard", "Raj", "Amy", "Bernadette"]
4
5  # expand an existing list with .append()
6  >>> L2.append("Priya")
7  >>> print(L2)
8  ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
9
10 # concatenate L1 and L2 with .extend()
11 >>> L1.extend(L2)
12 >>> print(L1)
13 ["Leonard", "Penny", "Sheldon", "Howard", "Raj", "Amy", "Bernadette", "Priya"]
```

Can I change the order of a list's elements?

One of the most common list manipulation tasks consists of changing the order of an item's list. As shown in Snippet 4.14, it is possible to use the `.reverse()` method to reverse the elements of the list in place (see line 5) while sorting an item's list can be carried out with the `.sort()` method.

Snippet 4.14 — methods for changing list items in place

```
1  # create a list
2  >>> L = ["Howard", "Raj", "Amy", "Bernadette", "Priya"]
3
4  # reverse the list's item positions
5  >>> L.reverse()
6  >>> print(L)
7  ["Priya", "Bernadette", "Amy", "Raj", "Howard"]
8
9  # sort the list's items
10 >>> L.sort()
11 >>> print(L)
12 ["Amy", "Bernadette", "Howard", "Priya", "Raj"]
```

3.4 Dictionaries

What is a dictionary?

Along with lists, dictionaries are one of the most flexible built-in data types in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key* instead of by *positional offset*.

Why should I use a dictionary?

Dictionaries take the place of records, search tables, and any other sort of aggregation where item names are more meaningful than item positions.

What type of objects can I include in a dictionary?

Like lists, dictionaries can contain objects of any type, and they support nesting to any depth(they can contain lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

How do I create a dictionary?

Snippet 4.15 shows two different ways to create a dictionary. A dictionary can be created by including key-value pairs among curly braces (see line 2). In the example, there are three keys associated with Marvel characters and as many values, which can be thought of as the characters' position in an ideal power rank. A colon separates a key and its associated value. The second way to create a dictionary is based on Python's builtin `dict`, mapping key onto values, and `zip`, which iterates over two elements in parallel. Specifically, `zip` creates the one-to-one correspondence between keys (characters) and values (character's power) that is passed as the argument of `dict`. We will analyze the topic of iterations extensively in sections 3.10 and 3.11.

Snippet 4.15 — initializing a new dictionary object

```
1 # method 1
2 >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4 # method 2
5 >>> CHARACTERS = ["Captain Marvel", "Living Tribunal", "One-Above-All"]
6 >>> RANK = [3, 2, 1]
7 >>> D = dict(zip(CHARACTERS, RANK))
8 >>> print(D)
9 {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
```

How do I fetch a dictionary's values?

Dictionaries' items cannot be accessed via positional offsets — like lists. Instead, we fetch the individual items using the dictionary keys shown in Snippet 4.16 (see line 5). The reference key is passed among brackets. When the dictionary at hand contains nested dictionaries (see line 9), it is possible to concatenate multiple queries, namely, sequences of keys between brackets (see line 21).

Snippet 4.16 — fetching dictionary items

```
# the dictionary
>>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}

# let's fetch Captain Marvel's position in the Marvel characters' power rank
>>> D["Captain Marvel"]
3

# a dictionary of dictionaries
>>> D = {
        "Dr. Strange": {
                "first_appearance": 1963,
                "created_by": "Lee & Ditko"
                },
        "Iron Man": {
                "first_appearance": 1963,
                "created_by": "Lee, Lieber, Heck & Kirby"
                },
    }

# let us fetch the creator of Dr. Strange
>>> D["Dr. Strange"]["created_by"]
"Lee & Ditko"
```

Are dictionaries mutable?

Dictionaries, like lists, are mutable. Thus, we can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The del statement works here, too; it deletes the entry associated with the key specified as an index (see Snippet 4.17).

Snippet 4.17 — dictionary mutability examples

```
# the dictionary
>>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}

# let us change the power rank for Captain Marvel
>>> D["Captain Marvel"] = 12
>>> print(D)
{"Captain Marvel": 12, "Living Tribunal": 2, "One-Above-All": 1}

# let us eliminate the character Living Tribunal
>>> del D["Living Tribunal"]
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1}

# let us add a further character
>>> D["Wanda Maximoff"] = 4
>>> print(D)
{"Captain Marvel": 12, "One-Above-All": 1, "Wanda Maximoff": 4}
```

What are the most common methods to manipulate a dictionary?

Like for lists, Python offers many methods to manipulate dictionary objects. Table 3.8 reports some of the most common methods and synopses. The first three methods, `.keys()` `.values()` `.items()`, get the constitutive elements of dictionaries: keys, values, and key-value pairs, respectively. The fourth method, `.get(key, default?)` gets the value for a specific key. The fifth method, `.update()`, updates the value for a specific key. Like `.update()`, `.popitem()`, `.pop()`, and `d.clear()` alter the information of a dictionary in place. The first removes the value of a certain key; the second removes the item (a key-value pair) for a certain key; the latter deletes all dictionary items. Finally, `.copy()` creates a shallow copy of an existing dictionary.

TABLE 3.8
Popular Dictionary Methods

Method	Synopsis
`D.keys()`	Get all dictionary keys
`D.values()`	Get all dictionary values
`D.items()`	Get all dictionary key-value pairs as tuples
`D.get(key, default?)`	Query a dictionary element by key
`D.update(D2)`	Update a dictionary key's value
`D.popitem()`	Remove the value corresponding to a certain key
`D.pop(key, default?)`	Remove the item at the given position in the list,
`D.clear()`	Delete all dictionary items
`D.copy()`	Copy the target dictionary

How do I access the information in a dictionary?

Snippet 4.18 shows how to use built-in methods to carry out three fundamental tasks: accessing dictionary keys (see line 5), values (see line 9), and items (i.e., key-value pairs, see line 13). It is worth noticing that the three methods illustrated in the example yield specific dictionary objects such as `dict_keys`, `dict_values`, and `dict_items`. Translating one of these dictionary objects into a list — if needed — is straightforward (see line 17).

Snippet 4.18 — accessing the information included in a dictionary

```
1  # the dictionary
2  >>> D = {"Captain Marvel": 3, "Living Tribunal": 2, "One-Above-All": 1}
3
4  # get the keys
5  >>> D.keys()
6  dict_keys(["Captain Marvel", "Living Tribunal", "One-Above-All"])
7
8  # get the values
9  >>> D.values()
10 dict_values([3, 2, 1])
11
12 # get the items
13 >>> D.items()
14 dict_items([("Captain Marvel", 3), ("Living Tribunal", 2), ("One-Above-All", 1)])
15
16 # get the keys as a list
17 >>> list(D.keys())
18 ["Captain Marvel", "Living Tribunal", "One-Above-All"]
```

3.5 Tuples

What is a tuple?

Tuples are sequences of immutable Python objects. They are similar to lists, but they are immutable. Tuples are created by enclosing a comma-separated list of values in parentheses.

Are Tuples mutable?

Tuples are immutable, which means that once they are created, they cannot be changed!!

Why should I use tuples?

Tuples are useful for storing data that is not to be changed, such as the coordinates of a point in a two-dimensional space. In general, we use tuples any time information integrity is a concern — in other words when we want to ensure the information included in an object will not change because of another reference in our program.

How do I create a tuple?

Python objects, separated by a comma, must be included between parentheses (see Snippet 4.19, line 2).

How do I access the information in a tuple?

By positional offsets, like lists (see Snippet 4.19, lines 5 and 9).

Snippet 4.19 — creating and accessing a tuple

```
# the tuple
>>> T = ("Captain Marvel", 3)

# access a tuple element
>>> T[0]
"Captain Marvel"

# access a tuple element
>>> T[1]
3
```

Can I convert a tuple into a list?

Yes, you can. To do that, you must pass the tuple as the argument of `list` (see Snippet 4.20).

Snippet 4.20 — tuple conversion

```
>>> T = ("Captain Marvel", 3)

# from a tuple to a list
>>> L = list(T)
>>> print(L)
["Captain Marvel", 3]

# amend L's items
>>> L[1] = 4

# get back to a tuple
>>> T = tuple(L)
>>> print(T)
("Captain Marvel", 4)
```

Can I create an advanced data container based on a tuple?

`collections` is a module that is shipped with Python and provides data containers that are alternative to Python's general purpose built-in containers, i.e., `dict`, `list`, `set`, and `tuple`. One of these containers can be created with the function `namedtuple` (see Snippet 4.21), which allows annotating the tuple items with names. In line 2, we import the function `namedtuple` from the `collections` module. In line 5, we create an ad hoc class that best represents the structure of our sample data concerning Marvel characters' names and the year they first appeared in the comic series. The first argument taken by the function is customary and regards the name of the class we are about to create. The second argument is a list with the names of the attributes included in our data structure. In line 8, we use the newly created class `Rec` to create a tuple, which is eventually printed as per line 11.

Snippet 4.21 — creating an annotated tuple with the collection module

```
1  # import the named tuple function from the module collection
2  >>> from collections import namedtuple
3
4  # create an ad hoc class object 'Rec' that fits our data structure
5  >>> Rec = namedtuple("Rec", ["character", "first_appearance"])
6
7  # use the generated class "Rec"
8  >>> IRONMAN = Rec("Iron Man", 1963)
9
10 # A named-tuple record
11 >>> IRONMAN
12 Rec(character="Iron Man", first_appearance=1963)
```

3.6 Sets

What is a set?

A `set` is an *unordered* collection of *unique* and *immutable* objects.

What does it mean that sets are unordered collections?

By design, `set` is a data structure with *undefined element ordering* (see Snippet 4.22 — the outcome included in line 6 does not follow any particular order).

What does it mean that sets have unique items?

By definition, an item appears only once in a set, no matter how many times it is added (see Snippet 4.22, line 2 Vs. line 7).

Snippet 4.22 — creating a set

```
# create a list
>>> L = ["a", "a", "b", "c", "c"]

# get a set from L
>>> S = set(L)
>>> print(S)
{"b", "a", "c"}
```

Why should I use sets?

Sets made this way support common mathematical set operations (see Snippet 4.23). Hence, they have a variety of applications, especially in numeric and database-focused work.

Snippet 4.23 — set operations

```
# create two sets
>>> X = set(["a", "b", "c"])
>>> Y = set(["c", "d", "e"])

# set difference
>>> X - X
set()
>>> X - Y
{"a", "b"}

# union
>>> X | Y
{"a", "b", "c", "d", "e"}

# intersection
>>> X & Y
{"c"}

# superset
>>> X > Y
False

# subset
>>> X < Y
False
```

3.7 Files

How do the files in my operating system relate with Python?

Your Python program may involve input and/or output operations. In other words, you may want to read data from a file stored in your machine and/or write the outcome of your analysis to a file. The built-in function `open` creates a Python file object, which serves as a link to a file residing on your machine. As Lutz notes:

> *"Compared to the types you've seen so far, file objects are somewhat unusual. They are considered a core type because a built-in function creates them, but they're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks" (page 282)*

How do I open a file?

You open a pipe to a file using the built-in function `open`. The output of the function is a `file` object.

How do I source the data stored in a file?

You open a pipe to a file using the built-in function `open`. The output of the function is a `file` object. Snippet 4.24 illustrates how to use `open` for data sourcing. In the first part of the snippet, we create a `file` object to read the data included in the existing file `my_file.txt`.[11] At least, we have to pass one argument to `open`: the path pointing to the file. A second optional argument is `mode`, which specifies the mode in which the file is opened to the source. It defaults to `r`, which means open for reading in text mode. Other common values are `w` for writing,[12] `x` for exclusive creation, and `a` for appending. [13] To read a file's contents, we use the `.read()` method (see line 9), returning a string object (see line 10).

Snippet 4.24 — data input with open

```
1  # create a pipe to a file
2  >>> file = open(file="my_file.txt", mode="r")
3
4  # calling "file" yields the attributes of the file object
5  >>> file
6  <_io.TextIOWrapper name="my_file.txt" mode="r" encoding="UTF-8">
7
8  # let us source the data
9  >>> data = file.read()
10 >>> print(data)
11 Hi there
12
13 # close the pipe
14 >>> file.close()
```

How do I write the data in the current Python session to a file?

Snippet 4.25 illustrates how to use `open` for data writing. In the first part of the snippet, we create three strings — i.e., the information we are manipulating in the active Python session (see lines 2, 4, and 6). Then, we create a file object in 'writing' mode (see the value passed to `mode`, line 9). Finally, we manipulate the three strings (as a sample task, in line 12, we concatenate `FIRSTLAW`, `SECONDLAW`, `THIRDLAW`) and write the result to a file (line 16).

Snippet 4.25 — data output with open

```
1  # the strings (data) to save permanently to a file
2  >>> FIRSTLAW = "A robot may not injure a human being or, through inaction, "\
3                 "allow a human being to come to harm."
4  >>> SECONDLAW = "A robot must obey the orders given it by human beings except "\
5                 "where such orders would conflict with the First Law."
6  >>> THIRDLAW = "A robot must protect its own existence as long as such "\
7                 "protection does not conflict with the First or Second Law."
8
9  # create a pipe to a file
10 >>> file = open(file="my_file.txt", mode="w")
11
12 # concatenate the strings
13 >>> TO_WRITE = "\n".join([FIRSTLAW, SECONDLAW, THIRDLAW])
14
15 # write the concatenated strings
16 >>> file.write(TO_WRITE)
17
18 # close the pipe
19 >>> file.close()
```

How about reading a single line from a file?

Hold on: what is a line? A string whose last character is \n. We can read a single line from a file using the `.readline()` method (see Snippet 4.26). Such a method starts by reading the first line included in the file (see line 11); then, it reads any subsequent lines included in the file (see line 15); when it reaches the end of the file (EOF), it returns the empty string "" (see line 19).

Snippet 4.26 — reading one line at a time with `.readline()`

```
1  # the strings (data) to save permanently to a file
2  >>> DATA = "The first line\nThe second line"
3
4  # create a pipe to a file and write DATA
5  >>> file = open(file="my_file.txt", mode="w")
6  >>> file.write(DATA)
7  >>> file.close()
8
9  # read one line from the file
10 >>> file = open(file="my_file.txt", mode="r")
11 >>> file.readline()
12 "The first line\n"
13
14 # calling file.readline() again reds the subsequent line
15 >>> file.readline()
16 "The second line"
17
18 # ... and so on until the end of the file is reached
19 >>> file.readline()
20 ""
```

How about reading multiple lines at a time?

The `.readlines()` method reads the lines from a file and returns them as a list (see Snippet 4.27).

Snippet 4.27 — reading multiple lines at a time with `.readlines()`

```
# the strings (data) to save permanently to a file
>>> DATA = "A\nB\nC\nD"

# create a pipe to a file and write DATA
>>> file = open(file="my_file.txt", mode="w")
>>> file.write(DATA)
>>> file.close()

# read multiple lines
>>> file = open(file="my_file.txt", mode="r")
>>> file.readlines()
['A\n', 'B\n', 'C\n', 'D']
```

What are the most common file methods?

Table 3.9 illustrates some key file methods' names and their corresponding synopsis.

TABLE 3.9
Popular File Methods

Method	Description
`file.close()`	Closes the file
`file.detach()`	Returns the separated raw stream from the buffer
`file.fileno()`	Returns a number that represents the stream as per the OS' perspective
`file.flush()`	Flushes the internal buffer
`file.isatty()`	Returns whether the file stream is interactive or not
`file.read()`	Returns the file content
`file.readable()`	Returns whether the file stream can be read or not
`file.readline()`	Returns one line from the file
`file.readlines()`	Returns a list of lines from the file
`file.seek()`	Change the file position
`file.seekable()`	Returns whether the file allows us to change the file position
`file.tell()`	Returns the current file position
`file.truncate()`	Resizes the file to a specified size
`file.writable()`	Returns whether the file can be written to or not
`file.write()`	Writes the specified string to the file
`file.writelines()`	Writes a list of strings to the file

Notes: `file` is a fictionary object used to illustrate the usage of the file methods.

3.8 Python Statements and Syntax

What is a Python statement?

In his popular book 'Learning Python,' Lutz provides a concise and effective description of what a Python statement is:

> *In simple terms, statements are the things you write to tell Python what your programs should do. If, as suggested [omitted], programs "do things with stuff," then statements are the way you specify what sort of things a program does. Less informally, Python is a procedural, statement-based language; by combining statements, you specify a procedure Python performs to satisfy a program's goals.*

What are the most common Python statements?

Table 3.10 illustrates common Python statements, their role, and application examples. Some of these statements were used in the examples considered so far. Other statements — the majority — will be faced in the next sections of the current chapter and/or the subsequent chapters.

TABLE 3.10
Python Statements

Statement	Role	Example
`import`	Module access	`import math`
`from`	Attribute access	`from math import sqrt`
`class`	Building ad hoc objects	`class Subclass(Superclass):  def method(self):  pass`
`del`	Deleting references	`del a`
Assignment	Creating references	`a = "before b"`
Calls and other expressions	Running functions	`file.write("Hello")`
`print`	Printing objects	`print("Hello")`
`if/elif/else`	Selecting actions	`if "abc" in text:  print(text)`
`for/else`	Iteration	`for x in mylist:  print(x)`
`while/else`	General loops	`while X > Y: print("Hello")`
`pass`	Empty placeholder	`while True:  pass`
`break`	Loop exit	`while True:  if exit test():  break`
`continue`	Loop continue	`while True:  if skiptest():  continue`
`def`	Functions and methods	`def f(a, b, c=1, *d):  print(a+b+c+d[0])`
`return`	Functions results	`def f(a, b, c=1, *d):  return a+b+c+d[0]`
`yield`	Generator functions	`def gen(n):  for i in n:  yield i*2`

3.9 Control Flow (or If-Then Statements)

What is control flow in Python?

Many Python statements we write are compound statements: there is one statement nested inside another. The outer statement is called the 'if' statement, and the inner statement is called the 'then' statement. The 'if' statement determines whether to execute the 'then' statement. Specifically, the 'then' statement is executed insofar as the 'if' statement evaluates to 'True.' Snippet 4.28 illustrates a control flow case, a simple rule-based product recommender that suggests products based on users' purchasing patterns. If product `x` belongs to a user's set of past purchases, then a certain item is recommended; otherwise, no recommendation is offered (see lines 7 and 8, containing the `else` statement).

Snippet 4.28 — an example of control flow

```
1 # a set with a customer's past purchases
2 >>> S = set(["a", "x", "u"])
3
4 # a rule-based product recommender
5 >>> if "x" in S:
6 ...     print("Customers who bought x also bought Air Jordan 7 Retro Miro")
7 ... else:
8 ...     pass
9 Customers who bought x also bought Air Jordan 7 Retro Miro
```

Does 'end of the line' mean 'end of the statement'?

Any Python statements are contained in the same line — the end of the line equates to the statement end. In the interest of redundancy, Python statements do not traverse multiple lines. In Snippet 4.28, the if statement is in line 5; the then statement is in line 6.

How do I create a nested statement?

The colon character is required to separate the if statement from the 'then' statement (see Snippet 4.28 line 5).

Does indentation have substantive meaning in Python?

Yes, it does. 'Then' statements are indented (with a tab or four consecutive spaces). Do not creatively use indents to embellish your code — that is not consistent with Python's rules and design principles (see Snippet 4.28 line 6).

Does 'end of indentation' mean 'end of nested statements'?

'Then' statements are indented (with a tab or four consecutive spaces). In Snippet 4.28, the indentation in line 6 makes lines 5 and 6 to be evaluated together.

How do I concatenate multiple 'then' statements?

Exmple 4.29 shows how to use `elif` to concatenate multiple 'if-then' statements in the same control flow. Like in Snippet 4.28, the 'else' statement defines the residual behavior of the control flow; that is, what Python does when both the 'if' and 'elif' statements evaluate to 'False.'

Snippet 4.29 — an example of control flow with multiple 'if-then' statements

```
1  # a list with a customer's past purchases
2  >>> S = set(["a", "w", "u"])
3
4  # a rule-based product recommender
5  >>> if "x" in S:
6  ...     print("Customers who bought x also bought Air Jordan 7 Retro Miro")
7  ... elif "w" in S:
8          print("How about Converse Chuck Taylor All Star?")
9  ... else:
10         print("Falling short of suggestions --- I'm a dull recommender!")
11 How about Converse Chuck Taylor All Star?
```

How do I create nested if-then statements?

In Python, it is possible to nest an if-then statement into another. In Snippet 4.30, the 'if' statements in lines 6 and 8 are nested inside the 'if' statement in line 5. It is worth noticing that lines 7 and 9 — regarding 'then' statements — are indented twice because they terminate distinct if-then statements nested in the broader if-then statement commencing on line 5.

Snippet 4.30 — an example of nested control flow

```
1  # a list with a customer's past purchases
2  >>> S = set(["a", "x", "b"])
3
4  # a rule-based product recommender
5  >>> if "x" in S:
6  ...     if "a" in S:
7  ...         print("Customers who bought x & a also bought Air Force")
8  ...     elif "u" in S:
9  ...         print("Customers who bought x & u also bought Air Max 95")
10 ... else:
11 ...     print("Falling short of suggestions --- I'm a dull recommender!")
12 Customers who bought x & a also bought Air Force
```

3.10 While and For Loops

Loops?!

Oftentimes, we write Python statements that repeat the same task — i,e. they loop a certain number of times or over multiple items.

How do I write loops in Python?

Using `for` and `while` statements.

What is the difference between `for` and `while` statements?

The `while` statement provides a way to code general loops. The `for` statement is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

Snippet 4.31 — `while` loop examples

```
# loop until reaching a numeric threshold
>>> i = 0
>>> while i <= 3:
...     print(i)
...     i = i + 1
0
1
2
3

# loop until an empty string is returned
>>> x = "Indiana Jones"
>>> while x != "":
...     print(x)
...     x = x[1:]
...
Indiana Jones
ndiana Jones
diana Jones
iana Jones
ana Jones
na Jones
a Jones
 Jones
Jones
ones
nes
es
s
```

Can you give an example of a `while` loop?

`while` statements run a code block insofar as a test evaluates to True. In the upper section of Example 4.31, we assign a `i` to a number. Then we create a for loop with the following elements: the first one is a statement testing whether `i` is smaller or equal to 3 (see line 3); the second element is the loop body (indented), which is repeated as long as the test evaluates to `True`. It is worth noticing that every iteration of the loop body produces a unitary increase in `i` — the program leaves the loop after four iterations. In the lower section of Snippet 4.31, we assign the variable `x` to a string (line 13), which we eventually print (line 14) and slice (line 15) until we get an empty string (line 13x).

Can you give an example of a `for` loop?

`for` steps through a sequence of items and carries out a task. In the upper section of Snippet 4.32, we print the result of a mathematical operation deployed over the items of a `list` (an example of a Python iterable object). The code included in line 2 assigns the variable `item` to an element of iterable temporarily. Then, the code block (indented) is executed over the temporary object. In the lower section of Snippet 4.32, the execution of the loop operates a mathematical expression over the items of a first list and appends the outcome to a second list (line 10).

Snippet 4.32 — a `for` loop example

```
1  # print the result of a mathematical operation carried out over a list of items
2  >>> for item in [0, -99, 13, 6.54]:
3  ...     print(item ** 0.5)
4  0.0
5  (6.092540900222253e-16+994987437106662j)
6  3605551275463989
7  2.5573423705088842
8  # run a mathematical operation on a list of items and append the outcome
9  # to a second list
10 >>> input = [2, 8, 1]
11 >>> output = []
12 >>> for item in input:
13 ...     output.append(item + 1)
14 >>> print(output)
15 [3, 9, 2]
```

How do I use `for` loops with dictionaries?

Like lists, dictionaries are iterable objects. In the upper section of Snippet 4.33, we create a dictionary and iterate over its items printing a simple predicate. As we know from section 3.4, we access a dictionary's values by keys. Hence, in line 5, we retrieve the keys of `D`. Then, in line 12, we fetch the value of the temporary object `k`, namely, `D[k]`. Particularly, we print the temporary object `k`, the string object `IS`, and the value associated with `k`; that is, `D[k]`. In Snippet 34, we accomplish the same task of Snippet 33. However, the loop regards a dictionary's items — i.e., key-value pairs — instead of keys (that is self-evident from the comparison of Snippet 33's line 55 and Snippet 34's line 5).

Snippet 4.33 — looping on dictionary keys

```
1  # the dictionary
2  >>> D = {"Thor": "Asgardian", "Vision": "android", "Wanda Maximoff": "human"}
3
4  # get the keys of D
5  >>> keys = D.keys()
6  >>> print(keys)
7  dict_keys(['Thor', 'Vision', 'Wanda Maximoff'])
8
9  # iterate over the keys to fetch the dictionary values and do something
10 # with them
11 >>> for k in keys:
12 ...     print(k + " IS " + D[k])
13 Thor IS Asgardian
14 Vision IS android
15 Wanda Maximoff IS human
```

Snippet 4.34 — looping on dictionary items

```
1  # the dictionary
2  >>> D = {"Thor": "Asgardian", "Vision": "android", "Wanda Maximoff": "human"}
3
4  # get the items of D
5  >>> items = D.items()
6
7  # iterate over key-value pairs and do something with them
8  >>> for k, v in items:
9  ...     print(k + " IS " + v)
10 Thor IS Asgardian
11 Vision IS android
12 Wanda Maximoff IS human
```

Why are counter `for` loops so popular?

The built-in class `range` provides an immutable sequence that is particularly helpful for loops that repeat an action a certain number of times. Snippet 35 shows an example of a for loop with `range`.

Snippet 4.35 — a counter loop example

```
1  # show the outcome of range
2  >>> list(range(3))
3  [0, 1, 2]
4
5  # use range in a for loop
6  >>> for i in range(3):
7  ...     print(i, ":-)")
8  0 :-)
9  1 :-)
10 2 :-)
```

What is a nested `for` loop?

A Python statement that contains multiple for loops is a nested `for` loop. Mainly, a `for` loop allows to jointly carry out a task over the elements of two iterables. The outer loop considers the individual items of the first iterable (see Snippet 4.26, line 6); the inner loop (indented) considers the individual items of the second list (see line 7). Once we have created a pair of temporary objects, we can do something with it (see line 8).

Snippet 4.36 — a nested loop example

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # create all permutations of letters and colors and print them
6  >>> for i in LETTERS:
7  ...     for j in COLORS:
8  ...         print(i, " - ", j)
9  x  <->  blue
10 x  <->  green
11 x  <->  red
12 y  <->  blue
13 y  <->  green
14 y  <->  red
15 z  <->  blue
16 z  <->  green
17 z  <->  red
```

Nested `for` loops Vs. `zip` for loops?

Contrarily to the nested `for` loops, which considers all permutations containing multiple iterables' items, the built-in zip steps through several iterables *in parallel*, producing tuples with an item from each one. As shown in Snippet 4.37, there is neither an inner nor an outer for loop in this case — instead, there is a single loop considering two temporary objects, `i` and `j`, that occupy the same position in the offset of the iterables at hand (see line 6; the first item from the first iterable goes with the first item from the second iterable, the second item from the first iterable goes with the second item from the second iterable, and so on).

Snippet 4.37 — looping over two iterables in parallel with `zip`

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # create one-to-one matches of items and do something with them
6  >>> for i, j in zip(LETTERS, COLORS):
7  ...     print(i, " <-> ", j)
8  x  <->  blue
9  y  <->  green
10 z  <->  red
```

3.11 Iterations and Comprehensions

Is there any other Python iterator besides `while` and `for` loops?

As we know from the previous section, `while` and `for` loops can handle most repetitive tasks programs need to perform. However, Python provides additional tools to make loops *easier to write/read* and *more efficient*. One of the most prominent tools is `list comprehension`.

Why do I use `list comprehensions`?

To create a list containing the outcome of an action repeated over an iterable's items (see Snippet 4.38, line 14).

How do I create `list comprehensions`?

We include a Python statement containing a `for` clause among brackets (see Snippet 4.38, line 12).

Snippet 4.38 — for loop Vs. list comprehension

```
1  # the for loop way
2  # --+ create an empty list
3  L = []
4  # --+ create a for loop appending the square of some items
5  >>> for i in range(3):
6  ...     L.append(i ** 2)
7  # --+ print the list
8  >>> print(L)
9  [0, 1, 4]
10
11 # the list comprehension way
12 >>> L = [i ** 2 for i in range(3)]
13 >>> print(L)
14 [0, 1, 4]
```

How do I implement a nested `for` loop in list comprehensions?

As Snippet 4.39 shows, a nested for loop becomes a one-liner in a list comprehension. The first `for` clause in line 7 would correspond to the outer `for` loop reported in Snippet 4.36, whereas the second `for` clause in line 7 would correspond to the inner `for` loop reported in Snippet 4.36.

Snippet 4.39 — nested for loop with list comprehensions

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # implementing a nested for loop with a list comprehension
6  >>> LETTER2COLOR = ["{} <-> {}".format(i, j) for i in LETTERS for j in COLORS]
7  ['x <-> blue',
8   'x <-> green',
9   'x <-> red',
10  'y <-> blue',
11  'y <-> green',
12  'y <-> red',
13  'z <-> blue',
14  'z <-> green',
15  'z <-> red']
```

Can I use the `zip` generator within a list comprehension?

Yes, we can. To do that, the `for` clause must consider two iterables simultaneously (see Snippet 4.40, line 6).

Snippet 4.40 — looping over two iterables in parallel with `zip` and list comprehension

```
1  # the lists
2  >>> LETTERS = ["x", "y", "z"]
3  >>> COLORS = ["blue", "green", "red"]
4
5  # implementing a nested for loop with a list comprehension
6  >>> LETTER2COLOR = ["{} <-> {}".format(i, j) for i, j in zip(LETTERS, COLORS)]
7  ['1 <-> blue', '2 <-> green', '3 <-> red']
```

Can I embed control flow in a list comprehension?

Yes, we can. To do that, the `for` clause must be preceded by an `if` statement and, at least, an `else` statement (see for example Snippet 4.40's line 22).

Snippet 4.41 — control flow in list comprehensions

```
1  # import the function log from math
2  from math import log
3
4  # the object to manipulate
5  >>> L1 = [0, 1, 2]
6
7  # the for loop way
8  # --+ the empty list
9  L2 = []
10 # --+ the for loop appending the log of some items
11 >>> for i in L1:
12 ...     if i > 0:
13 ...         L2.append(log(i))
14 ...     else:
15 ...         L2.append(log(i + 0.001))
16 # --+ print the list
17 >>> print(L2)
18 [-6.907755278982137, 0.0, 0.6931471805599453]
19
20 # the list comprehension way
21 # --+ the list comprehension is a one-liner!
22 >>> L2 = [log(i) if i > 0 else log(i + 0.001) for i in L1]
23 # --+ print the list
24 >>> print(L2)
25 [-6.907755278982137, 0.0, 0.6931471805599453]
```

Notes

[7]Lutz, Mark. *Learning Python: Powerful object-oriented programming.* O'Reilly Media, Inc., 2013.
[8]Floating numbers are stored in binaries with an assigned level of precision typically equivalent to 15 or 16 decimals.
[9]As per the documentation of the Python programming language, `math` cannot be used with complex numbers.
[10]The official Python documentation has an extensive section on operator precedence rules in the section dedicated to syntax of expressions
[11]For the sake of simplicity, we assume the target file is located in the same directory as the Python script.
[12]By default, `w` truncates the file if it already exists
[13]If encoding is not specified, the encoding used is platform-dependent. Specifically, `locale.getpreferredencoding(False)` is called to get the current locale encoding. Character encoding assigns

numbers to graphical characters, especially the written characters of human language, allowing them to be stored, transmitted, and transformed using digital computers.

Chapter 4

Technical & Scientific Computation with NumPy

At the end of the chapter, you will be able to:

- Create efficient data structures called `ndarrays`
- Populate `ndarrays` with simulated or real-world data
- Manipulated `ndarrays` with a variety of scalable routines
- Carry out exploratory data analysis with NumPy
- Read and write `ndarrays` in many formats

4.1 Installing NumPy

Does NumPy come with the official Python installation file?

No, it does not. You need to install it separately using the package manager `pip`.

I am an Anaconda user: do I need to install NumPy 'separately'?

No, you do not. NumPy is included in 'base,' the default environment of Anaconda. However, if you create a new environment, you need to install NumPy — and all the other modules you need — with the package manager `conda`.

How do I install NumPy?

The easiest way is using the command line. Anaconda users run `$ conda install numpy scipy`, whereas Python official release users run `$ pip install numpy scipy`. Anaconda users can also install the modules they need from within Anaconda-Navigator.

4.2 NumPy ndarray

What is a NumPy **ndarray**?

Put simply, an `ndarray` is a data container, like dictionaries and lists.

Can **ndarrays** contain objects of different type?

No, they cannot. An **ndarray** must contain homogenous items; that is, items of the same type.

How do I create an **ndarray**?

As shown in Snippet 5.1, we pass an object to `numpy.array`. If the object we pass is a scalar, a 0-dimensional array containing object is returned (line 5). Passing a list to `numpy.array` produces a one-dimensional array (line 9); passing a list of lists produces a two-dimensional array (line 13); finally, passing a list of lists of lists produces a three-dimensional array (line 18).

Snippet 5.1 — creating an ndarray

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # a 0-D array
5  >>> np.array(0)
6  array(0)
7
8  # a 1-D array
9  >>> np.array([1, 2, 3, 4])
10 array([1, 2, 3, 4])
11
12 # a 2-D array
13 >>> np.array([[1, 2], [3, 4]])
14 array([[1, 2],
15        [3, 4]])
16
17 # a 3-D array
18 >>> np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
19 array([[[1, 2],
20         [3, 4]],
21
22        [[5, 6],
23         [7, 8]]])
```

What are the distinctive features of `ndarrays`?

`ndarrays` have been designed — and tuned over time — with flexibility and efficiency in mind. For example, `ndarrays` allows to carry out computations on arrays with a syntax similar to scalar values. As shown in Snippet 5.2, we can multiply an array by a scalar (line 10) and add two vectors (14). That is not possible if we use pure Python code. Multiplying a list by a scalar N replicates the ordered collection of items N times (see line 23). Adding two lists yields concatenation (see line 30). At the same time, `ndarrays` support the analysis of large volumes of data[14].

Snippet 5.2 — NumPy allows to manipulate vectors with an expressive syntax

```
1  # import numpy with the socially accepted alias 'np'
2  import numpy as np
3
4  # generate some random
5  >>> DATA = np.random.randn(3)
6  >>> print(DATA)
7  [-0.44144029 -0.44451097  0.31997294]
8
9  # can we multiply a list by a scalar with NumPy? Of course!
10 >>> print(DATA * 3)
11 [-4.4144029 , -4.44510974,  3.19972941]
12
13 # can we sum two arrays with NumPy? Of course!
14 >>> print(DATA + DATA)
15 [-0.88288058, -0.88902195,  0.63994588]
16
17 # let us try to replicate the previous tasks in pure Python?
18 # --+ get the DATA as a list
19 >>> DATA = list(DATA)
20 >>> print(DATA)
21 [-0.4414402896845323, -0.4445109735283278, 0.31997294069261617]
22 # --+ is the NumPy syntax of line 10 still valid if we use a list? Nope
23 >>> print(DATA * 3)
24 [
25 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
26 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
27 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617
28 ]
29 # --+ is the NumPy syntax of line 14 still valid if we use a list? Nope
30 >>> print(DATA + DATA)
31 [
32 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617,
33 -0.4414402896845323, -0.4445109735283278, 0.31997294069261617
34 ]
```

How do I check an `array`'s number of dimensions?

In Snippet 5.3, we saw NumPy infers an `ndarrays`'s number of dimensions from the data. Snippet 5.3 shows how to access `.ndim`, the `ndarrays`'s attribute concerning the number of dimensions. In the example, `DATA` has two dimensions (e.g., coordinates).

How do I check an `array`'s shape?

The lower section of Snippet 5.3 shows how to access `.shape`, the `ndarrays`'s attribute concerning the shape. In the example, each dimension of the `DATA` has size 2.

Snippet 5.3 — an array's number of dimensions and shape

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the data
>>> DATA = np.array([[1, 2], [3, 4]])
>>> print(DATA)
[[1 2]
 [3 4]]

# get the number of dimensions
>>> DATA.ndim
2

# get the shape
>>> DATA.shape
(2, 2)
```

What are the attributes of an `array`?

Table 4.1 illustrates the common use attributes of `ndarrays`.

TABLE 4.1
Common Use Attributes of NumPy `array`

Attribute	Synopsis
`DATA.flags`	Information about the memory layout of the array
`DATA.shape`	Tuple of array dimensions
`DATA.strides`	Tuple of bytes to step in each dimension when traversing an array
`DATA.ndim`	Number of array dimensions
`DATA.data`	Python buffer object pointing to the start of the array's data
`DATA.size`	Number of elements in the array
`DATA.itemsize`	Length of one array element in bytes
`DATA.nbytes`	Total bytes consumed by the elements of the array
`DATA.dtype`	Data-type of the array's elements

Notes: `DATA` is a fictionary object used to illustrate the usage of the `array` attributes.

I know that a NumPy array must contain homogenous data — but which object types are allowed?

Table 4.2 reports the NumPy data types. Python beginners are not supposed to appreciate the distinctive attributes of each type. Instead, they may want to get a clear understanding of the high-level types, namely, floating points, complex, integer, boolean, string, or general Python objects. When working on sophisticated projects requiring more control over the storage types, it is highly suggested to get a thorough knowledge of the types in Table 4.2. It is worth noticing that dtypes are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases, they map directly onto an underlying disk or memory representation, making it easy to read and write binary data streams to disk and connect to code written in a low-level language like C or Fortran. The numerical `dtypes` are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating-point value takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`.

How do I specify the data type of an `array`?

As Snippet 5.4 shows, dtype is an optional argument of `ndarray`. It is also possible to change `dtype` using the method `.astype()`.

Snippet 5.4 — specifying a nd changing dtype

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# accept the default type
>>> A = np.array([1, 2, 3, 4, 5])

# check the type
>>> A.dtype
dtype('int64')

# specify the type
>>> A = np.array([1, 2, 3, 4, 5], dtype=np.int32)

# check the type
>>> A.dtype
type('int32')

# type change
>>> S = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
>>> S = S.astype(float)
>>> S.dtype
dtype('float64')
```

TABLE 4.2
NumPy Data Types

Type	Type Code	Synopsis
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point float32 f4 or f Standard single-precision floating-point; compatible with C float
float64	f8 or d	Standard double-precision floating-point; compatible with C double and Python
float object	float128 f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

4.3 Array Creation Routines

Does NumPy offer recipes for creating arrays?

Yes, it does. NumPy has seven families of array-creating routines:

- From shape or value
- From existing data
- Creating record arrays (`np.rec`)
- Numerical ranges
- Building matrices
- The Matrix class

Creating arrays from shape or value

This family of routines creates arrays with a certain number of dimensions, shapes, values, and attributes. Snippet 5.5. shows how to create:

- an array with a certain shape and a constant scalar (see lines 5 — `.zeros`, 12 — `.ones`, 19 — `full`), and
- an array containing an identity matrix (see lines 26 — `.eye` — and 33 — `.identity`)

Snippet 5.5 — creating arrays from shape or value

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# create an array with zeros only
>>> np.zeros([4,4])
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])

# create an array with ones only
>>> np.ones((4,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])

# create a full matrix with a given scalar
>>> np.full((4,4), -99)
array([[-99, -99, -99, -99],
       [-99, -99, -99, -99],
       [-99, -99, -99, -99],
       [-99, -99, -99, -99]])

# create an identity array of a given shape with .eye
>>> np.eye(4, 3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])

# create an identity array with .identity
>>> np.identity(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

TABLE 4.3
Routines for Creating Arrays from Shape or Value

Routine	Synopsis
`np.empty(shape[, dtype, order, like])`	Return a new array of given shape and type, without initializing entries
`np.empty_like(prototype[, dtype, order, subok, ...])`	Return a new array with the same shape and type as a given array
`np.eye(N[, M, k, dtype, order, like])`	Return a 2-D array with ones on the diagonal and zeros elsewhere
`np.identity(n[, dtype, like])`	Return the identity array
`np.ones(shape[, dtype, order, like])`	Return a new array of given shape and type, filled with ones
`np.ones_like(a[, dtype, order, subok, ...])`	Return an array of ones with the same shape and type as a given array
`np.zeros(shape[, dtype, order, like])`	Return a new array of given shape and type, filled with zeros
`np.full(shape, fill_value[, dtype, order, like])`	Return a new array of given shape and type, filled with fill value
`full_like(a, fill_value[, dtype, order, ...])`	Return a full array with the same shape and type as a given array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Creating arrays from existing data

In Snippets 5.1 – 5.4, we saw how to use `array` for passing data to a NumPy array. Snippet 5.6 shows other routines to create NumPy arrays from existing data, including for example:

- `.fromfunction`, creating an array by executing a function over each coordinate (line 8)
- `.fromfile`, creating an array from data in a text or binary file (line 19)
- `.loadtxt`, loading data from a text file (line 37). The example represents a real-world data set containing both numeric and text information. The first argument we pass `.loadtxt` is a file object. To correctly parse the data, we also pass the following discretionary arguments to `.loadtxt`: i) `comments="#"` indicates that any lines in the file commencing with `#` must be considered a comment, not a piece of data; ii) `delimiter=","` indicates that two items separated by the character `,` belong to different fields (i.e., 'columns' to use a spreadsheet-alike vocabulary); iii) textttquotechar="'' indicates that strings are enclosed between double quotes[15].

Snippet 5.6 — creating arrays from existing data

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # get data from a function
5  # --+ create a function
6  >>> my_function = lambda x, y: x - 0.5 * y ** 2
7  # --+ create an array from my for given coordinates
8  >>> np.fromfunction(my_function, (3, 3), dtype=float)
9  array([[ 0. , -0.5, -2. ],
10        [ 1. ,  0.5, -1. ],
11        [ 2. ,  1.5,  0. ]])
12
13 # get data from a binary file
14 # --+ create an array from a list of numbers
15 >>> D = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9]))
16 # --+ save the raw data to a binary file
17 >>> D.tofile("data.bin")
18 # --+ read the data back
19 >>> np.fromfile("data.bin", dtype=int)
20
21 # get data from a text file
22 # --+ create a string with the data and some qualitative comments on them
23 >>> S = """
24 # Below are some demographic data about Michael J. Jordan (basketball player)
```

```
# from Wikipedia.
#
# Data labels are:
#
# NAME, BORN, NBA CHAMPIONSHIPS, AVERAGE POINT PER GAME
"Jordan, Michael Jeffrey","17-02-1963",6,30.1
"""
# --+ write the data to a file
>>> with open("my_data", "w") as pipe:
...     pipe.write(S)
>>> pipe.close()
# --+ read the data and assign them to a NumPy array
>>> np.loadtxt(
...   open("my_data", "r"),
...       dtype={
...             "names": (
...                       "NAME",
...                       "BORN",
...                       "NBA CHAMPIONSHIPS",
...                       "AVERAGE POINT PER GAME"
...                           ),
...             "formats": ("S30", "S10", "i1", "f2"),
...             },
...       comments="#",
...       delimiter=",",
...       quotechar='"'
...   )
array((b'Jordan, Michael Jeffrey', b'17-02-1963', 6, 30.1),
       dtype=[('NAME', 'S30'),('BORN', 'S10'),
             ('NBA CHAMPIONSHIPS', 'i1'),
             ('AVERAGE POINT PER GAME', '<f2')]
)
```

TABLE 4.4
Routines for Creating Arrays from Existing Data

Routine	Synopsis
`np.array(object[,dtype, copy, subok, ...])`	Create an array
`np.asarray(a[, dtype, order, like])`	Convert the input to an array
`np.asanyarray(a[, dtype, order, like])`	Convert the input to an ndarray, but pass ndarray subclasses through
`np.ascontiguousarray(a[, dtype, like])`	Return a contiguous array (ndim ¿= 1) in memory (C order)
`np.asmatrix(data[, dtype])`	Interpret the input as a matrix
`np.copy(a[, order, subok])`	Return an array copy of the given object
`np.frombuffer(buffer[, dtype, count, offset, like])`	Interpret a buffer as a 1-dimensional array
`np.fromfile(file[, dtype, count, sep, offset, like])`	Construct an array from data in a text or binary file
`np.fromfunction(function, shape, *[, dtype, like])`	Construct an array by executing a function over each coordinate
`np.fromiter(iter, dtype[, count, like])`	Create a new 1-dimensional array from an iterable object
`np.fromstring(string[, dtype, count, like])`	A new 1-D array initialized from text data in a string
`np.loadtxt(fname[, dtype, comments, delimiter, ...])`	Load data from a text file

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Record arrays

NumPy arrays do not contain any information about the attributes of the data. For example, a NumPy array cannot accommodate any meta-data, such as the fields' names in the data. Here is where `.rec` kick in (see Table 4.5). For example, .core.records.array allows to flexibly specify a field's type and name (see Snippet 5.7, line 8). Once a 'recarray' is created, it is possible to fetch its data by field name (see Snippet 5.7, line 11).

Snippet 5.7 — creating record arrays

```
1  # import records array with an alias that does not conflict with
2  # `standard' NumPy arrays
3  >>> from numpy.core.records import array as recarray
4
5  # the data
6  >>> LOCS = [("51.5072° N", "0.1276° W"), ("35.6762° N", "139.6503° E")]
7
8  # create a recarray
9  >>> D = recarray(LOCS, formats=["U12", "U12"], names=["Latitude", "Longitude"])
10
11 # fetch the data by field name
12 >>> D.Latitude
13 array(['51.5072° N', '35.6762° N'], dtype='<U12')
```

TABLE 4.5
Routines for Creating Record Arrays

Routine	Synopsis
`np.core.records.array(obj[, dtype, shape, ...])`	Construct a record array from a wide variety of objects
`np.core.records.fromarrays(arrayList[, dtype, ...])`	Create a record array from a (flat) list of arrays
`np.core.records.fromrecords(recList[, dtype, ...])`	Create a recarray from a list of records in text form
`np.core.records.fromstring(datastring[, dtype, ...])`	Create a record array from binary data
`np.core.records.fromfile(fd[, dtype, shape, ...])`	Create an array from binary file data

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Creating numerical ranges

One may want to create a numerical range for different reasons, including running functional analysis or computer simulation. NumPy has a bunch of array-creating routines for numerical ranges (see Table 4.6), some of which are quite popular in technical and scientific computation as well as data science. For example, `np.arange` and `np.linspace` frequently appear in Python programs when it comes to create evenly spaced values in a certain interval and evenly spaced samples respectively (see Snippet 5.8, lines XX and XX). Another popular routine is `.meshgrid`, returning coordinate matrices from coordinate vectors.

Snippet 5.8 — creating numerical ranges

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# two ranges of evenly spaced values
# --+ evenly spaced values between 0 and 10
>>> np.arange(0, 10, 1)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# --+ ... equivalent to
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
# --+ evenly spaced values between 0 and 10 divided by a 2-unit step
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])

# 50 evenly spaced values between 0 and 1
>>> np.linspace(0, 1, 10)
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])

# get coordinate matrices from coordinate vectors
# --+ the 'x-' and 'y-axis' vectors
>>> X = np.linspace(0, 1, 10)
>>> Y = np.linspace(0, 1, 5)
# --+ get 'x-axis' ('y-axis') coordinates for any value of vector Y (X)
>>> XX, YY = np.meshgrid(X, Y)
>>> XX
array([[0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ],
       [0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
        0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ]])
```

```
>>> YY
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25],
       [0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 , 0.5 ],
       [0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75],
       [1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  ]])
# --+ create a matrix from X and Y
>>> ZZ = np.sqrt(XX**2 + YY**2)
# --+ check the dimensions of the newly created objects
>>> print(XX.shape, YY.shape, ZZ.shape)
>>> ((101, 101), (101, 101), (101, 101))
# --+ make a contour plot showing the associations among X, Y, and Z
# (see Figure 5.1)
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax = plt.contourf(X, Y, ZZ)
>>> plt.axis('scaled')
>>> plt.colorbar()
>>> plt.show()
```

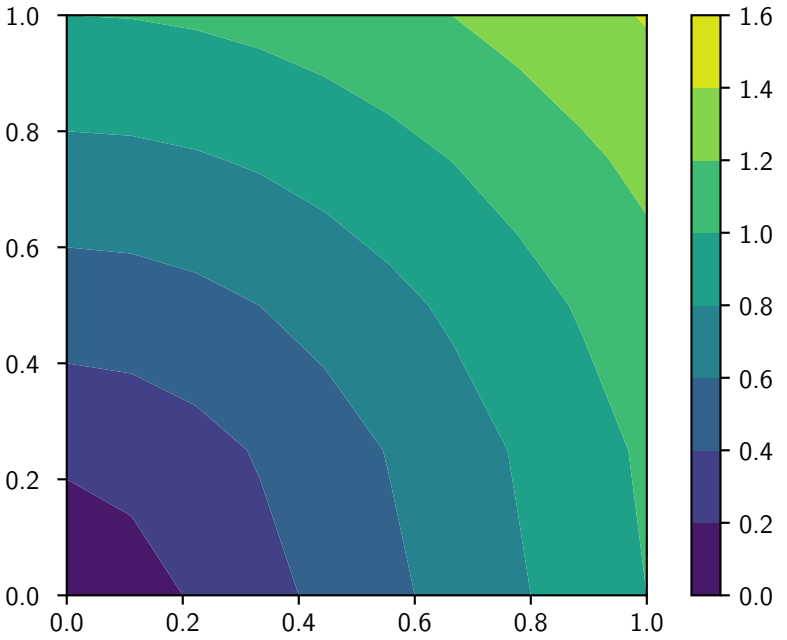


Figure 4.1: A contour plot showing the associations among X, Y, and Z
Notes. — the data behind this plot come from Snippet 5.8, lines 22 - 44.

TABLE 4.6
Routines for Numerical Ranges

Routine	Synopsis
`np.arange([start,] stop[, step,][, dtype, like])`	Return evenly spaced values within a given interval
`np.linspace(start, stop[, num, endpoint, ...])`	Return evenly spaced numbers over a specified interval
`np.logspace(start, stop[, num, endpoint, base, ...])`	Return numbers spaced evenly on a log scale
`np.geomspace(start, stop[, num, endpoint, ...])`	Return numbers spaced evenly on a log scale (a geometric progression)
`np.meshgrid(*xi[, copy, sparse, indexing])`	Return coordinate matrices from coordinate vectors
`np.mgrid`	`nd_grid` instance which returns a dense multi-dimensional "meshgrid"
`np.ogrid`	`nd_grid` instance which returns an open multi-dimensional "meshgrid"

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Building matrices

NumPy has routines to create arrays from an existing matrix as well as build matrices with certain properties (see Table 4.7) As Snippet 5.9 shows, `.diag` creates an array by fetching a matrix's diagonal (see line 10), while `.tri` creates a triangular matrix (see line 17).

Snippet 5.9 — creating arrays from existing data

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # create an array by fetching a matrix diagonal
5  # --+ the matrix
6  >>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
7  >>> M.shape
8  (3, 3)
9  # --+ the new array
10 >>> A = np.diag(M)
11 >>> print(A)
12 [1 5 9]
13 >>> A.shape
14 (3,)
15
16 # create a triangular matrix
17 >>> np.tri(10, 10)
18 array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
19        [1., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
20        [1., 1., 1., 0., 0., 0., 0., 0., 0., 0.],
21        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
22        [1., 1., 1., 1., 1., 0., 0., 0., 0., 0.],
23        [1., 1., 1., 1., 1., 1., 0., 0., 0., 0.],
24        [1., 1., 1., 1., 1., 1., 1., 0., 0., 0.],
25        [1., 1., 1., 1., 1., 1., 1., 1., 0., 0.],
26        [1., 1., 1., 1., 1., 1., 1., 1., 1., 0.],
27        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

TABLE 4.7
Routines for Building Matrices

Routine	Synopsis
`np.diag(v[, k])`	Extract a diagonal or construct a diagonal array
`np.diagflat(v[, k])`	Create a two-dimensional array with the flattened input as a diagonal
`np.tri(N[, M, k, dtype, like])`	An array with ones at and below the given diagonal and zeros elsewhere
`np.tril(m[, k])`	Lower triangle of an array
`np.triu(m[, k])`	Upper triangle of an array
`np.vander(x[, N, increasing])`	Generate a Vandermonde matrix

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

The Matrix Class

In Snippet 5.9, line 17, I claim to create a matrix. The outcome displayed in line 18 is consistent with the concept of 'matrix' we have been taught in a typical linear algebra class: what a human being sees is a set of numbers arranged in rows and columns. However, in NumPy terms, the object displayed in line 18 is an array with two dimensions. If we want to create a NumPy Matrix Class object[16], we have to call `np.matrix`, a subclass of `np.ndarray`. Table 4.8 illustrates the two matrix-creating routines of NumPy.

What is the advantage of using a matrix class object?

As we will see later on in this chapter, the advantage of using a matrix class object is that we can use the matrix class object to perform matrix operations with a simple and intuitive syntax. For example, we can use the matrix class object to perform matrix manipulation multiplication, addition, and subtraction. In the lower section of Snippet 5.10, we use `.linalg.inv` to compute the inverse of a matrix.

Snippet 5.10 — creating a NumPy Matrix Class object

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # create a matrix class object
5  # --+ the arrays to populate a matrix class object
6  >>> A = np.array([0, 1, 2])
7  >>> B = np.array([-99, 203, 1009])
8  >>> C = np.array([-1000, -1001, -1002])
9  # --+ the matrix class object
10 >>> M = np.matrix([A, B, C])
11 matrix([[    0,     1,     2],
12         [  -99,   203,  1009],
13         [-1000, -1001, -1002]])
14
15 # get the inverse of M
16 >>> np.linalg.inv(M)
17 matrix([[-1.60040278e+00,  1.98412698e-03, -1.19642857e-03],
18         [ 2.19880556e+00, -3.96825397e-03,  3.92857143e-04],
19         [-5.99402778e-01,  1.98412698e-03, -1.96428571e-04]])
```

TABLE 4.8
Routines for the Matrix Class

Routine	Synopsis
`np.mat(data[, dtype])`	Interpret the input as a matrix
`bmat(obj[, ldict, gdict])`	Build a matrix object from a string, nested sequence, or array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

4.4 Array Manipulation Routines

I have an array. What can I do with it?

NumPy offers many array manipulation routines (see Table A.7), which can be grouped around the following families:

- Basic operations
- Changing array shape
- Transpose-like operations
- Changing number of dimensions
- Changing kind of array
- Joining arrays
- Splitting arrays
- Tiling arrays
- Adding and removing elements
- Rearranging elements

Presenting every NumPy array manipulation routine would require writing a dedicated book (and certainly falls beyond the remit of an introductory module on Python). That said, let me whet the reader's appetite by illustrating a sample of miscellaneous routines (see Snippet 5.11). Here is the sequence of tasks we accomplish: first, we create an array of shape `(6,)` (see line 5); then, we change the shape of the array (line 10) and transpose its rows and columns (line 16); in the lower section of the snippet we join two arrays using several routines: `.concatenate` joins two arrays along the desired axis (the default axis is the first one, meaning the second arrays are concatenated row-wise); `.vstack` requires to pass a tuple of arrays to stack row-wise; `.hstack` requires to pass a tuple of arrays to stack column-wise. It is worth noticing that the rules for vector/matrix manipulation we learned at school apply to the routines that join arrays. In other words, two NumPy arrays can be joined if and only have the same length on the joining axis. For example, it is possible to stack vertically two arrays with the shape `(3, 3)` and `(2, 3)` because they have the same number of columns.

Snippet 5.11 — miscellaneous array-manipulating routines

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the array
>>> A = np.arange(6)
>>> A
array([0, 1, 2, 3, 4, 5])

# reshape the array into a 2x3 array
>>> A = A.reshape(2, 3)
>>> A
array([[0, 1, 2],
       [3, 4, 5]])

# transpose the array
>>> A.T
array([[0, 3],
       [1, 4],
       [2, 5]])

# get back to a 'flat' array with shape (6,)
>>> A.ravel()
array([0, 1, 2, 3, 4, 5])

# reshape the array into a 3x2 array
>>> A.reshape(3, 2)
array([[0, 1],
       [2, 3],
       [4, 5]])

# join two arrays --- note the dimensions for the concatenation must match
>>> np.concatenate((A.reshape(3, 2), A.T))
array([[0, 1],
       [2, 3],
       [4, 5],
       [0, 3],
       [1, 4],
       [2, 5]])

# stack two arrays vertically (ROW-WISE)
>>> np.vstack((A, np.array([6, 7, 8])))

# stack two arrays horizontally (COLUMN-WISE)
>>> np.hstack((A, np.array([6, 7]).reshape(2, 1)))
array([[0, 1, 2, 6],
       [3, 4, 5, 7]])
```

4.5 Universal Functions in NumPy

What is a universal function?

A universal function, or `ufunc`, is a function that performs element-wise operations on `ndarrays`. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results[17].

What is the rationale for using `ufuncts`?

Per the previous point, using a `ufunc` offer substantial performance advantages *vis a' vis* non-vectorized code — i.e., code using built-in Python iterators.

What are the `unfunct` options available in NumPy?

There are circa sixty universal functions implemented in NumPy. For the sake of convenience, the full list of `ufunc` options is reported in Tables A.3 - A.6. The following sections of the current chapter will illustrate how to use some of the popular central `ufuncts` in NumPy.

4.6 Mathematical Functions

What are the mathematical functions available in NumPy?

There are many mathematical functions available in NumPy, some of which are implemented as universal functions (for a complete list of universal functions, see Tables A.3). The available routines can be grouped into:

- Trigonometric functions
- Hyperbolic functions
- Rounding functions
- Sums, products, and differences
- Exponential and logarithmic functions
- Extrema finding

What are the mathematical functions available in NumPy?

Similar to the case of array-manipulating routines, a detailed discussion of every NumPy mathematical function exceeds the scope of these notes. However, every function can be accessed using the same procedure; we need a set of values to pass to the function's argument- that is it! Snippet 5.12 shows the procedure to use a NumPy mathematical functions by means of two trigonometric functions, namely, `.sin` and `.cos`. In line 9, we create a range of values to assign to the variable `X`; then, in line 11, we create two further arrays assigned to the outcome of `.sin` and `.cos` respectively; finally, we use the Matplotlib module to visualize the two functions (if you do not get the Matplotlib code logic, do not worry at all — you will familiarize yourself with it in the Fall Term module 'SMM635, Data Visualization.')

Snippet 5.12 — NumPy trigonometric functions in action

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # import a data viz module
5  >>> import matplotlib.pyplot as plt
6
7  # trigonometric functions
8  # --+ x-values
9  >>> X = np.arange(0, 2 * np.pi, 0.# --+ plot SI
10 # --+ y-values
11 >>> SI, CS = np.sin(X), np.cos(X)
12
13 # plot the functions
14 # --+ plot SI
15 >>> fig = plt.figure(figsize=(2.5, 2.5))
16 >>> ax = fig.add_subplot(111)
17 >>> ax.axhline(y=0, color="k", linewidth=0.5)
18 >>> ax.set_xticks([0, 0.5 * np.pi, np.pi, 1.5 * np.pi, 2 * np.pi])
19 >>> ax.set_xticklabels(
20 ...     ["0", r"$\frac{1}{2} \pi$", r"$\pi$", r"$\frac{3}{2} \pi$", r"$2 \pi$"]
21 ... )
22 >>> plt.xlabel("$X$")
23 >>> plt.ylabel("$sine(X)$")
24 >>> ax.grid(True)
25 >>> ax.plot(X, SI, color="Blue")
26 >>> plt.title("A")
27 >>> plt.show()
28 # --+ plot CS
29 >>> fig = plt.figure(figsize=(2.5, 2.5))
30 >>> ax = fig.add_subplot(111)
31 >>> ax.axhline(y=0, color="k", linewidth=0.5)
32 >>> ax.set_xticks([0, 0.5 * np.pi, np.pi, 1.5 * np.pi, 2 * np.pi])
33 >>> ax.set_xticklabels(
```

```
...     ["0", r"$\frac{1}{2} \pi$", r"$\pi$", r"$\frac{3}{2} \pi$", r"$2 \pi$"]
... )
>>> plt.xlabel("$X$")
>>> plt.ylabel("$cosine(X)$")
>>> ax.grid(True)
>>> ax.plot(X, CS, color="Blue")
>>> plt.title("B")
>>> plt.show()
```

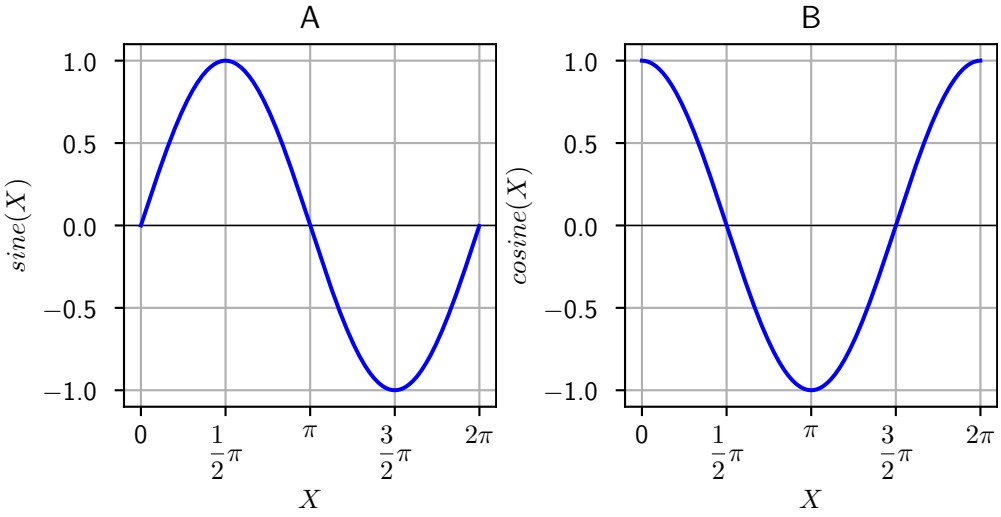


Figure 4.2: Visual representation of NumPy `.sin` and `.cos` functions as per Snippet 5.12.

4.7 Statistics

What are the statistical functions available in NumPy?

NumPy offers essential statistical functions sufficient to implement an Exploratory Data Analysis/descriptive statistics. Specifically, there are four families of statistical routines (see Tables 4.9, 4.10, 4.11, and 4.12):

- Order statistics (e.g., quantiles)
- Average and variances
- Correlations
- Histograms

Can I run multivariate statistical analysis with NumPy

Short answer: no. There are dedicated Python modules to run multivariate analyses, though. For example linearmodels and statsmodels are two popular modules to carry out econometric models in Python; scikit-learn is the acclaimed module for machine learning in Python.

TABLE 4.9
NumPy Statistical Routines: Order Statistics

Routine	Synopsis
`np.ptp(a[, axis, out, keepdims])`	Range of values (maximum-minimum) along an axis
`np.percentile(a, q[, axis, out, ...])`	Compute the q-th percentile of the data along the specified axis
`np.nanpercentile(a, q[, axis, out, ...])`	Compute the qth percentile of the data along the specified axis, while ignoring nan values
`np.quantile(a, q[, axis, out, overwrite_input, ...])`	Compute the q-th quantile of the data along the specified axis
`np.nanquantile(a, q[, axis, out, ...])`	Compute the qth quantile of the data along the specified axis, while ignoring nan values

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.10
numPy Statistical Routines: Average and Variances

Routine	Synopsis
`np.median(a[, axis, out, overwrite_input, keepdims])`	Compute the median along the specified axis
`np.average(a[, axis, weights, returned, keepdims])`	Compute the weighted average along the specified axis
`np.mean(a[, axis, dtype, out, keepdims, where])`	Compute the arithmetic mean along the specified axis
`np.std(a[, axis, dtype, out, ddof, keepdims, where])`	Compute the standard deviation along the specified axis
`np.var(a[, axis, dtype, out, ddof, keepdims, where])`	Compute the variance along the specified axis
`np.nanmedian(a[, axis, out, overwrite_input, ...])`	Compute the median along the specified axis, while ignoring NaNs
`np.nanmean(a[, axis, dtype, out, keepdims, where])`	Compute the arithmetic mean along the specified axis, ignoring NaNs
`np.nanstd(a[, axis, dtype, out, ddof, ...])`	Compute the standard deviation along the specified axis, while ignoring NaNs
`np.nanvar(a[, axis, dtype, out, ddof, ...])`	Compute the variance along the specified axis, while ignoring NaNs

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.11
NumPy Stastistical Routines: Correlating

Routine	Synopsis
`np.corrcoef(x[, y, rowvar, bias, ddof, dtype])`	Return Pearson product-moment correlation coefficients
`np.correlate(a, v[, mode])`	Cross-correlation of two 1-dimensional sequences
`np.cov(m[, y, rowvar, bias, ddof, fweights, ...])`	Estimate a covariance matrix, given data and weights

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.12
NumPy Statistical Routines: Histograms

Routine	Synopsis
`np.histogram(a[, bins, range, normed, weights, ...])`	Compute the histogram of a dataset
`np.histogram2d(x, y[, bins, range, normed, ...])`	Compute the bi-dimensional histogram of two data samples
`np.histogramdd(sample[, bins, range, normed, ...])`	Compute the multidimensional histogram of some data
`np.bincount(x, /[, weights, minlength])`	Count the number of occurrences of each value in an array of non-negative ints
`np.histogram_bin_edges(a[, bins, range, weights])`	Function to calculate only the edges of the bins used by the histogram function
`np.digitize(x, bins[, right])`	Return the indices of the bins to which each value in the input array belongs

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

So, how do I produce a set of summary stats in NumPy?

Snippet 5.13 shows how to create a typical set of summary stats, including an array's mean, standard deviation, minimum, maximum, and some percentiles of interest. Mainly, the snippet has three steps. First, we create an array (line 8). Second, we assign a couple of variables to NumPy statistical functions such as `.mean`, `.std`, `.min`, `np.max`, and `.percentile` (see lines 11-17). It is worth noting that `min` and `max` are keywords reserved for Python built-in functions. To avoid any name conflict and potential sources of confusion, in lines 13 and 14, we use the names `min_` and `max_`. Finally, we create a table displaying the variables created in the previous step (see lines 21, 23, and 27).

Snippet 5.12 — NumPy trigonometric functions in action

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # import a module for arranging numbers in a tabular format
5  >>> import tabulate
6
7  # the array
8  >>> X = np.array([0, 0, -3, 12, 7, 2, -4, 6, 9, -1, 5, 3, -1, 3, 10, 9])
9
10 # get the descriptive stats
11 >>> mean = np.mean(X)
12 >>> std = np.std(X)
13 >>> min_ = np.min(X)
14 >>> max_ = np.max(X)
15 >>> pp25 = np.percentile(X, 25)
16 >>> pp50 = np.percentile(X, 50)
17 >>> pp75 = np.percentile(X, 75)
18
19 # arrange the stats in a tabular format
20 # --+ create the table header
21 >>> headers = [
22         "Mean", "St. Dev.", "Min", "Max", "25th pp", "50th pp", "75th pp"
23         ]
24 # --+ format the floating point numbers to two decimal places and get a string
25 >>> stats = [
26         str(np.round(i, 3)) for i in [mean, std, min_, max_, pp25, pp50, pp75]
27         ]
28 # --+ print the table
29 >>> print(tabulate([stats], headers=headers, tablefmt="grid"))
30 +--------+------------+-------+-------+-----------+-----------+-----------+
31 |   Mean |   St. Dev. |   Min |   Max |   25th pp |   50th pp |   75th pp |
32 +========+============+=======+=======+===========+===========+===========+
33 |  3.562 |      4.756 |    -4 |    12 |     -0.25 |         3 |       7.5 |
34 +--------+------------+-------+-------+-----------+-----------+-----------+
```

Can I calculate Pearson's correlation coefficients in NumPy?

Yes, you can. Snippet 5.13 shows how to do so by using `.corrcoef`, one of the functions included in Table 4.11.

Snippet 5.13 — a matrix with Pearson's correlation coefficients

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
X = np.array([0, 0, -3, 12, 7, 2, -4, 6, 9, -1, 5, 3, -1, 3, 10, 9])
Y = np.array([12, 12, 4, 3, 9, 2, -6, 15, 0, -12, 15, -3, -1, 0, 0, 1])

# get Pearson's correlation coefficients
>>> np.corrcoef(X, Y)
array([[1.        , 0.19763628],
       [0.19763628, 1.        ]])
```

4.8 Linear Algebra

Is NumPy a good choice for linear algebra?

Yes, it is. NumPy offers a rich set of routines comparable to Matalab's one.[18] Currently, there are five families of routines concerning the field of linear algebra at large:

- products
- decomposition
- eigenvalues
- norms
- equations and inversions

For an overview of numPy's linear algebra routines, see Tables 4.13, 4.14, 4.15, 4.16, and 4.17.

TABLE 4.13
NumPy Linear Algebra Routines: Matrix and Vector Products

Routine	Synopsis
`np.dot(a, b[, out])`	Dot product of two arrays
`np.linalg.multi_dot(arrays, *[, out])`	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order
`np.vdot(a, b, /)`	Return the dot product of two vectors
`np.inner(a, b, /)`	Inner product of two arrays
`np.outer(a, b[, out])`	Compute the outer product of two vectors
`np.matmul(x1, x2, /[, out, casting, order, ...])`	Matrix product of two arrays
`np.tensordot(a, b[, axes])`	Compute tensor dot product along specified axes
`np.einsum(subscripts, *operands[, out, dtype, ...])`	Evaluates the Einstein summation convention on the operands
`np.einsum_path(subscripts, *operands[, optimize])`	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays
`np.linalg.matrix_power(a, n)`	Raise a square matrix to the (integer) to the power n
`np.kron(a, b)`	Kronecker product of two arrays

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.14
NumPy Linear Algebra Routines: Decompositions

Routine	Synopsis
`np.linalg.cholesky(a)`	Cholesky decomposition
`np.linalg.qr(a[, mode])`	Compute the qr factorization of a matrix
`np.linalg.svd(a[, full_matrices, compute_uv, ...])`	Singular Value Decomposition

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.15
NumPy Linear Algebra Routines: Matrix Eigenvalues

Routine	Synopsis
`np.linalg.eig(a)`	Compute the eigenvalues and right eigenvectors of a square array
`np.linalg.eigh(a[, UPLO])`	Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix
`np.linalg.eigvals(a)`	Compute the eigenvalues of a general matrix
`np.linalg.eigvalsh(a[, UPLO])`	Compute the eigenvalues of a complex Hermitian or real symmetric matrix

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.16
NumPy Linear Algebra Routines: Norms and Other Number

Routine	Synopsis
`np.linalg.norm(x[, ord, axis, keepdims])`	Matrix or vector norm
`np.linalg.cond(x[, p])`	Compute the condition number of a matrix
`np.linalg.det(a)`	Compute the determinant of an array
`np.linalg.matrix_rank(A[, tol, hermitian])`	Return matrix rank of array using SVD method
`np.linalg.slogdet(a)`	Compute the sign and (natural) logarithm of the determinant of an array
`np.trace(a[, offset, axis1, axis2, dtype, out])`	Return the sum along diagonals of the array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

TABLE 4.17
NumPy Linear Algebra Routines: Solving Equations and Inverting Matrices

Routine	Synopsis
`np.linalg.solve(a, b)`	Solve a linear matrix equation, or system of linear scalar equations
`np.linalg.tensorsolve(a, b[, axes])`	Solve the tensor equation a x = b for x
`np.linalg.lstsq(a, b[, rcond])`	Return the least-squares solution to a linear matrix equation
`np.linalg.inv(a)`	Compute the (multiplicative) inverse of a matrix
`np.linalg.pinv(a[, rcond, hermitian])`	Compute the (Moore-Penrose) pseudo-inverse of a matrix
`np.linalg.tensorinv(a[, ind])`	Compute the 'inverse' of an N-dimensional array

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Matrix and vector products

Snippet 5.14 shows how to use NumPy to get dot, inner, and outer products — whose definitions are reported in equations 4.1, 4.2, and 4.3 respectively:

$$x \cdot y = \sum_{i=1}^{n} x_i y_i \tag{4.1}$$

Where x and y are vectors of length n.

$$\langle x, y \rangle = x^T y \tag{4.2}$$

Where x^T is the transpose of x.

$$x \otimes y = x y^T \tag{4.3}$$

Where y^T is the transpose of y.

Snippet 5.14 — matrix and vector products

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
>>> X = np.arange(1, 11, 1)
>>> X
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

>>> Y = np.arange(10, 0, -1)
>>> Y
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])

# dot product
>>> np.dot(X, Y)
220

# inner product
# --+ let's reshape the arrays
>>> X = X.reshape(5, 2)
>>> X
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
>>> Y = Y.reshape(5, 2)
>>> Y
array([[10,  9],
       [ 8,  7],
       [ 6,  5],
       [ 4,  3],
       [ 2,  1]])
>>> np.inner(X, Y)
array([[ 28,  22,  16,  10,   4],
```

```
       [ 66,  52,  38,  24,  10],
       [104,  82,  60,  38,  16],
       [142, 112,  82,  52,  22],
       [180, 142, 104,  66,  28]])

# outer product
>>> np.outer([0, 1, 2], [4, 5, 6, 7])
array([[ 0,  0,  0,  0],
       [ 4,  5,  6,  7],
       [ 8, 10, 12, 14]])
```

...

Solving a system of equations

Snippet 5.15 shows how to solve the following system of equations:

$$\begin{cases} 4x + 2y = 2000 \\ 7x + 13y = 3000 \end{cases}$$

Snippet 5.15 — solving a system of equations such as

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the system of equations
# --+ left-hand side
>>> a = np.array([[4, 2], [7, 13]])
# --+ right-hand side
>>> b = np.array([2000, 3000])

# solve the system of equations
>>> np.linalg.solve(a, b)
array([526.31578947, -52.63157895])
```

How do I carry out a least-square estimation in NumPy?

The least-square estimator is a popular choice for estimating the relationship between two variables, one of the most common tasks in statistics and econometrics.[19] Specifically, the least-square estimator is defined as follows:

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{4.4}$$

Where $\hat{\beta}$ is the estimate of the regression coefficients, X is the matrix of regressors, y is the outcome vector, and $X^T X$.

In Snippet 5.16, we create two arrays, `x` — playing the role of the regressor — and `y` — which we pretend to be the outcome variable. Then, we arrange the data in an array containing `x` and a vector of ones. In this way, we add an intercept to the model, namely a scalar that does not change across the observations in the data (see line 11). Finally, we estimate the two regression coefficients of interest:

- The regression coefficient for `x`, capturing the expected change in `y` for a unitary increase in `x`
- The regression coefficient for the intercept, namely, the exepected value of `y` when the association between `x` and `y` is partialled-out

Let me stress that `.linalg.lstsq` returns four pieces of information:

- An array with the estimated regression coefficients
- The sums of squared residuals (i.e., the differences between the observed `y` and the values predicted by the model)
- The rank of the matrix containing the regressors
- The singular values of the matrix containing the regressors

Thus, in line 28, we fetch the first element of the returned array, i.e., the regression coefficients.

Figure 4.3 provides a visual representation of the least-square estimator.

Snippet 5.16 — computing the least-square solutions to a linear matrix equation

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the arrays
>>> x = np.arange(0, 15, 1)
>>> y = [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
and use the least-square estimator to e
# data preparation / adding a constant to the model
>>> A = np.vstack([x, np.ones(len(X))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 2.,  1.],
       [ 3.,  1.],
       [ 4.,  1.],
       [ 5.,  1.],
       [ 6.,  1.],
       [ 7.,  1.],
       [ 8.,  1.],
       [ 9.,  1.],
       [10.,  1.],
       [11.,  1.],
       [12.,  1.],
       [13.,  1.],
       [14.,  1.]])

# estimate the regression coefficients (a.k.a., the regression slopes) of
# the linear model
>>> b = np.linalg.lstsq(A, y)[0]
>>> b
array([2.15714286, 2.23333333])
```

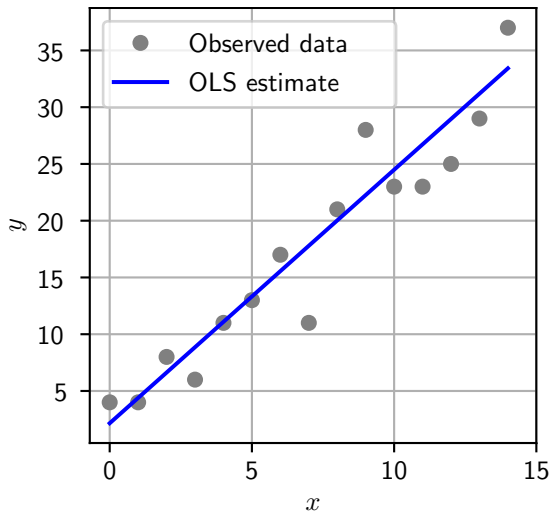


Figure 4.3: A visual illustration of the least-square estimation carried out in Snippet 5.16

4.9 Pseudorandom Number Generation

Why do I need pseudorandom number generators?

Psedudorenadom number generators play a central role in a computer simulation, a flexible and powerful tool that can be used in different ways, e.g., to get a better understanding of real-world data, to appreciate the functioning of complex systems, or for scenario analysis.[20]

What is the gamut of pseudorandom numbers available in NumPy?

NumPy has three families of pseudorandom number generators:

- Simple random data
- Permutations
- Distributions

Tables 4.18, 4.19, and 4.20 provide a summary of the available generators.

TABLE 4.18
NumPy Pseudorandom Generators: Simple Random Data

Routine	Synopsis
`integers(low[, high, size, dtype, endpoint])`	Return random integers from low (inclusive) to high (exclusive), or if endpoint=True, low (inclusive) to high (inclusive)
`random([size, dtype, out])`	Return random floats in the half-open interval [0.0, 1.0)
`choice(a[, size, replace, p, axis, shuffle])`	Generates a random sample from a given array
`bytes(length)`	Return random bytes

Notes: the statements included in the 'Routine' column assume `numpy.random.Generator` has been imported.

TABLE 4.19
NumPy Pseudorandom Generators: Permutations

Routine	Synopsis
`shuffle(x[, axis])`	Modify an array or sequence in place by shuffling its contents
`permutation(x[, axis])`	Randomly permute a sequence, or return a permuted range
`permuted(x[, axis, out])`	Randomly permute x along axis

Notes: the statements included in the 'Routine' column assume `numpy.random.Generator` has been imported.

TABLE 4.20
NumPy Pseudorandom Generators: Distributions

Routine	Synopsis
`beta(a, b[, size])`	Draw samples from a Beta distribution
`binomial(n, p[, size])`	Draw samples from a binomial distribution
`chisquare(df[, size])`	Draw samples from a chi-square distribution
`dirichlet(alpha[, size])`	Draw samples from the Dirichlet distribution
`exponential([scale, size])`	Draw samples from an exponential distribution
`f(dfnum, dfden[, size])`	Draw samples from an F distribution
`gamma(shape[, scale, size])`	Draw samples from a Gamma distribution
`geometric(p[, size])`	Draw samples from the geometric distribution
`gumbel([loc, scale, size])`	Draw samples from a Gumbel distribution
`hypergeometric(ngood, nbad, nsample[, size])`	Draw samples from a Hypergeometric distribution
`laplace([loc, scale, size])`	Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay)
`logistic([loc, scale, size])`	Draw samples from a logistic distribution
`lognormal([mean, sigma, size])`	Draw samples from a log-normal distribution
`logseries(p[, size])`	Draw samples from a logarithmic series distribution
`multinomial(n, pvals[, size])`	Draw samples from a multinomial distribution
`multivariate_hypergeometric(colors, nsample)`	Generate variates from a multivariate hypergeometric distribution
`multivariate_normal(mean, cov[, size, ..])`	Draw random samples from a multivariate normal distribution

Table 5.20 (cont'd)

Routine	Synopsis
`negative_binomial(n, p[, size])`	Draw samples from a negative binomial distribution
`noncentral_chisquare(df, nonc[, size])`	Draw samples from a noncentral chi-square distribution
`noncentral_f(dfnum, dfden, nonc[, size])`	Draw samples from the noncentral F distribution
`normal([loc, scale, size])`	Draw random samples from a normal (Gaussian) distribution
`pareto(a[, size])`	Draw samples from a Pareto II or Lomax distribution with a specified shape
`poisson([lam, size])`	Draw samples from a Poisson distribution
`power(a[, size])`	Draws samples in [0, 1] from a power distribution with positive exponent a - 1
`rayleigh([scale, size])`	Draw samples from a Rayleigh distribution
`standard_cauchy([size])`	Draw samples from a standard Cauchy distribution with mode = 0
`standard_exponential([size, dtype, method, out])`	Draw samples from the standard exponential distribution
`standard_gamma(shape[, size, dtype, out])`	Draw samples from a standard Gamma distribution
`standard_normal([size, dtype, out])`	Draw samples from a standard Normal distribution (mean=0, stdev=1)
`standard_t(df[, size])`	Draw samples from a standard Student's t distribution with df degrees of freedom
`triangular(left, mode, right[, size])`	Draw samples from the triangular distribution over the interval [left, right]
`uniform([low, high, size])`	Draw samples from a uniform distribution
`vonmises(mu, kappa[, size])`	Draw samples from a von Mises distribution
`wald(mean, scale[, size])`	Draw samples from a Wald, or inverse Gaussian, distribution
`weibull(a[, size])`	Draw samples from a Weibull distribution
`zipf(a[, size])`	Draw samples from a Zipf distribution

Notes: the statements included in the 'Routine' column assume `numpy.random.Generator` has been imported.

Permutation in NumPy

Permuting means changing the order of the elements in an array. Array permutations can be done 'in-place' (see Snippet 5.17, line 13), so the original array is modified, or a copy is created (see line 17). Note the outcome of the `np.random` functions are not reproducible. Simply, running the same NumPy generator n times might return n different outcomes. To ensure the reproducibility of NumPy code, we must initialize a random generator instance, as shown in the lower section of Snippet 5.17. Specifically, in line 23, we assign the object `rgn` to the outcome of `.default_rng`. The parameter passed to the generator, known as 'seed,' is an arbitrary number object. If you reproduce lines 23-25, you will get the outcome I got, displayed in line 26.

Snippet 5.17 — `.shuffle` Vs. `.permutation`

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # shuffling/permuting an existing array
5  # --+ the array
6  >>> A = np.arange(10)
7  >>> A
8  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
9  # --+ .shuffle produces in-place changes
10 >>> np.random.shuffle(A)
11 >>> A
12 array([6, 9, 3, 8, 7, 2, 4, 1, 0, 5])
13 # --+ .permutation creates a copy
14 >>> np.random.permutation(A)
15 array([7, 6, 5, 2, 1, 8, 0, 4, 9, 3])
16 >>> A
17 array([6, 9, 3, 8, 7, 2, 4, 1, 0, 5])
18
19 # shuffling an existing array and ensuring reproducibility
20 # --+ the array
21 >>> B = [0, 1, 2, 3]
22 # --+ initialize a random generator instance
23 >>> rng = np.random.default_rng(12345)
24 >>> rng.shuffle(B)
25 >>> B
26 [2, 0, 3, 1]
```

Sampling what?

Random sampling is the process of selecting a random subset of elements from an array. NumPy allows us:

- to sample the elements belonging to an existing array
- to sample the elements included in a certain interval
- to sample from a theoretical distribution

Lines 9 and 12 included in Snippet 5.18 show how to sample a given number of elements (see parameter `size`) from an existing array. The code in line 12 differs from the code in line 9 because of the optional parameter `replace` that is set to `False` (default is `True`). In so doing, an element can be sampled once and once only.

In lines 22 and 27, we create two arrays of shape `(10000,)` from the random normal (see `.random.normal`) and Poisson (see `.random.poisson`) distribution respectively. In the interesting of redundancy, `.random.normal` takes three mandatory arguments: the mean of the distribution (`loc`), the standard deviation of the distribution (`scale`), and the size of the array (`size`); (see `.random.poisson`) takes two mandatory arguments: the expected value/variance of the distribution (`lam`) and the size of the distribution (`size`). In lines 32 - 50, we use Matplotlib to visualize the distribution of the two arrays (see Figure 4.4)

Snippet 5.18 — `.integers` Vs. `.random` Vs. `.choice`

```
1  # import numpy with the socially accepted alias 'np'
2  >>> import numpy as np
3
4  # the array
5  >>> A = np.arange(10)
6
7  # random sampling from an existing array
8  # --+ sampling with replacement (items can be drawn multiple times)
9  >>> np.random.choice(A, size=7)
10 array([3, 5, 7, 3, 1, 5, 2])
11 # --+ sampling without replacement (items can be drawn only once)
12 >>> np.random.choice(A, size=7, replace=False)
13 array([4, 2, 6, 3, 1, 0, 5])
14
15 # sampling from a range
16 >>> np.random.integer(0, 10, size=4)
17 array([4, 3, 2, 3])
18
```

```
# sampling from a theoretical distribution
# --+ 10,000 items from a normal distribution with a mean of 10 and
# standard deviation of 10
>>> N = np.random.norm(loc=10, scale=10, size=10000)
>>> N
array([-1.80532765,  1.43318889,  1.14774224, ...,  0.38328412,
        1.37732456,  0.76801253])
# --+ 10,000 items from the Poisson distribution with lambda=10
>>> P = np.random.poisson(lam=10, size=10000)
>>> P
array([11,  7,  7, ...,  9,  6,  7])
# --+ visualize the two arrays
# ----+ Normal distribution
fig = plt.figure(figsize=(3, 3))
ax = fig.add_subplot(111)
ax.hist(N, color='blue', bins=50)
ax.set_ylabel('Count')
ax.set_xlabel('Value')
plt.title('Normal distribution data')
plt.grid(True,  ls="--")
plt.show()
# ----+ Poisson distribution
from collections import Counter
P_FR = Counter(P)
fig = plt.figure(figsize=(3, 3))
ax = fig.add_subplot(111)
ax.scatter(P_FR.keys(), P_FR.values(), color='blue')
ax.set_ylabel('Count')
ax.set_xlabel('Value')
plt.title('Poisson distribution data')
plt.grid(True, ls="--")
plt.show()
```

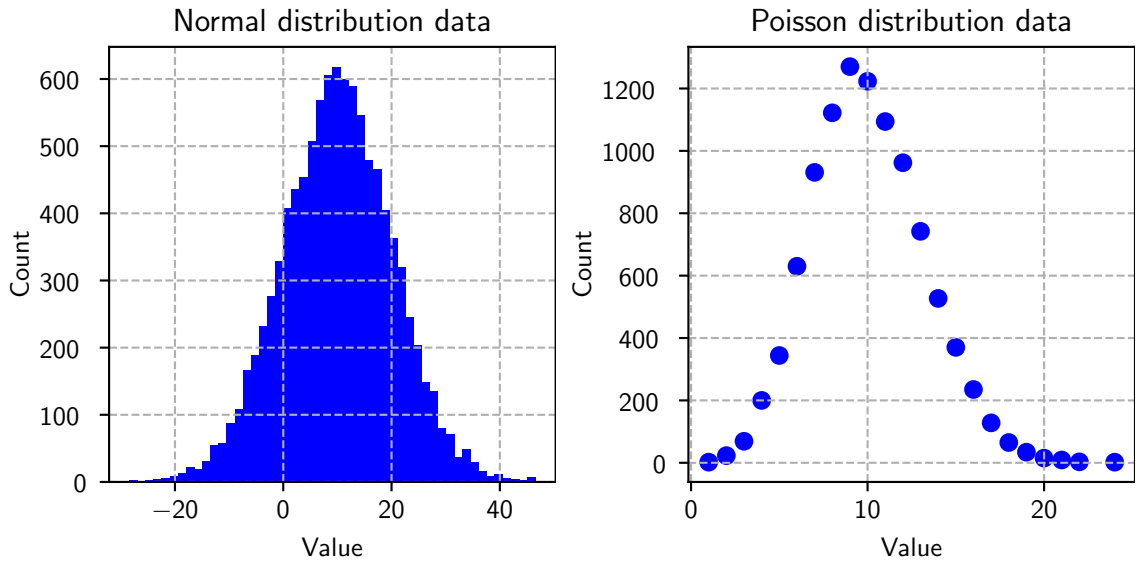


Figure 4.4: A visual illustration of the Normal and Poisson data generated in Snippet 5.18

4.10 File Input and Output (IO) with `ndarrays`

NumPy binary files

In Section 4.3, we saw how to create an array from the data included in a file stored locally. In this section, we will focus on how to input and output NumPy arrays as NumPy binary files

What are the advantages of IO with the `.npy` format?

There are several advantages:

- Efficiency! Efficiency!! Efficiency!!! IO with NumPy arrays is easy to code and fast to operate
- We do not have to care about character encoding aspects
- It is possible to wrap multiple arrays up in the same file

What are the IO routines for the `.npy` format?

There are four routines, summarized in Table 4.21.

Can you show me some IO examples?

Of course! Snippet 5.19 deals with the following cases:

- Single array case: lines 18 and 20 illustrate how to write and read an `.npy` file (accepting one and one array only!)
- Multiple array case: lines 35 - 60 illustrate how to write and read an `.npz` file (accepting multiple arrays)

Regarding the 'multiple array case,' it is worth noticing that it is possible to write the arrays to a file preserving the variable names assigned to the arrays (e.g., `A`, `B`, `C`, see line 55). Such an option is beneficial when the variable names are meaningful and, perhaps, we expect to load the data back sometime in the future, when we may or may not remember 'what is what.'

TABLE 4.21
File Input and Output with NumPy Arrays

Routine	Synopsis
`np.load(file[, mmap_mode, allow_pickle, ...])`	Load arrays or pickled objects from `.npy`, `.npz` or pickled files
`np.save(file, arr[, allow_pickle, fix_imports]))]`	Save an array to a binary file in NumPy `.npy` format
`np.savez(file, *args, **kwds)`	Save several arrays into a single file in uncompressed `.npz` format
`np.savez_compressed(file, *args, **kwds)`	Save several arrays into a single file in compressed `.npz` format

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Snippet 5.19 — IO with the .npy (.npz) format

```
# import numpy with the socially accepted alias 'np'
>>> import numpy as np

# the array
>>> A = np.reshape(np.arange(100), (10, 10))
>>> A
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],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])

# save A, delete it, and load the data back in
>>> np.save('A.npy', A)
>>> del A
>>> A = np.load('A.npy')
>>> A
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],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])

# working w/multiple arrays
# --+ save A and its transpose into a single file in uncompressed format
>>> np.savez('AAT.npz', A, A.T)
# --+ load the arrays back
# ----+ create a NpzFile object
>>> my_arrays = np.load('AAT.npz')
# ----+ check the arrays with the file attribute files
>>> my_arrays.files
['arr_0', 'arr_1']
# ----+ fetch the data on the second item
>>> my_arrays['arr_1']
array([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
       [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
       [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
       [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
       [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
       [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
       [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
```

```
       [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
       [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
       [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]])
# --+ it is also possible to save the arrays with their original names
>>> np.savez('AAT.npz', A=A, AT=A.T)
>>> my_arrays = np.load('AAT.npz')
>>> my_arrays.files
['A', 'AT']
>>> AT = my_arrays['AT']
>>> AT
array([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
       [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
       [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
       [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
       [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
       [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
       [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
       [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
       [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
       [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]])
```

Notes

[14]The Internet has many blog posts showing the performance of NumPy in linear algebra tasks is comparable to compiled languages, such as C.
[15]This feature was introduced with NumPy 1.23.
[16]The documentation of NumPy 1.23 states that *It is no longer recommended to use this class, even for linear algebra. Instead, use regular arrays. The class may be removed in the future.*
[17]Here is an interesting passage from the official NumPy documentation: *NumPy hands off array processing to C, where looping and computation are much faster than in Python. To exploit this, programmers using NumPy eliminate Python loops in favor of array-to-array operations. vectorization can refer both to the C offloading and to structuring NumPy code to leverage it.*
[18]Matlab is a numeric computing environment that is particularly popular in academia and industry.
[19]Students who want to familiarize with the field of econometrics are warmly encouraged to read Angrist, Joshua D., and Jörn-Steffen Pischke. Mostly harmless econometrics: An empiricist's companion. Princeton university press, 2009.
[20]The students who want to familiarize themselves with the role of computer simulation in understanding economics and social formations may want to refer to i) Axelrod, R. The complexity of cooperation, Princeton University Press, 1997; ii) Schelling, T. C. Micromotives and macrobehavior. WW Norton & Company, 1978; iii) Epstein, J. M. & Axtell, R. Growing Artificial Societies—Social Science from the Bottom Up. Artif Life 3, 237–242, 1997.

Chapter 5

Data Management with Pandas

At the end of the chapter, you will be able to:

- Create a DataFrame
- Manipulate the individual columns of a DataFrame
- Arrange a DataFrame in the way that best fits your data analysis needs
- Group, the cases of DataFrame, to perform aggregation and transformation tasks
- Work with multiple DataFrames
- Read and write data in a variety of formats

5.1 Pandas 101

What is special about Pandas?

Pandas is the *de facto* standard when it comes to managing data (in general, not only in Python!). So, what is special about Pandas?

- Compared to competing frameworks for data management,[21] Pandas offers a unique combination of features, including ease of use, flexibility, and performance
- Pandas has an edge in the area of time series[22] and panel data [23]— which is not surprising since Pandas was created for quantitative financial analysts, and it is still prevalent in the field of finance at large[24]
- Pandas builds on NumPy. That means access to a large variety of vectorized functions; that is, users/developers do not have to write boring and inefficient loops to perform operations on data
- The quality of the documentation covering Pandas is off-the-chart.[25]

Is Pandas a must-have skill and why?

Yes, it is. The reason is straightforward: data are clean and easy to work with in an ideal world. In the real world, data are messy and difficult to work with. So, it is important to have a tool that can help you work with data. Also, junior analysts are — often — required to carry out a substantial amount of data management. So you may want to get prepared before you land in the industry ...

Got it: Pandas is a must-have skill. But what can I achieve with it?

Data scientists and quantitative analysts use Pandas to carry out tasks falling into the following families:

- *Data preparation*, consisting of maximizing the quality of the data at hand; that is, getting information offering the most accurate representation of the business, economic, or financial process of interest. An example is cleaning the outcome of a web-crawling project targeting an online community of, say, beer enthusiasts
- *Data augmentation*, consisting of expanding on the raw information to create the variables that best capture the process we want to analyze. An example is creating a measure of customer satisfaction based on the data acquired using web-crawling
- *Data transformation*, consisting of arranging the data in the way that best supports a data visualization or analysis task. An example is converting a long data table, wherein observations are nested in one or multiple grouping variables, to a wide structure (hold your horses, we will see a concrete example later on in this chapter)

5.2 The Pandas DataFrame

What is a Pandas DataFrame?

Per the Pandas API, a DataFrame is a:

> *"two-dimensional, size-mutable, potentially heterogeneous tabular data"*

Put simply (i.e., from a user standpoint), a DataFrame is a tabular data structure with rows and columns. Typically, the rows are the cases (e.g., individuals, groups, firms, countries, etc.), and the columns are the so-called fields, variables, or features (e.g., wage, job satisfaction, stock-market value, etc.).[26] Figure 5.1 shows a stylized representation of a case-by-variable data structure, very common in Pandas.

Case	**Var a**	**Var b**	...	**Var i**	...	**Var k**
1	a_1	b_1	...	i_1	...	k_1
2	a_2	b_2	...	i_2	...	k_2
j	a_j	b_j	...	i_j	...	k_j
n	a_n	b_n	...	i_n	...	k_n

Figure 5.1: A stylized representation of a case-by-variable data structure

How do I create a DataFrame?

Mainly, there are two alternatives:

- Option 1: passing tables loaded onto the current Python session[27] to the .DataFrame class
- Option 2: sourcing external data, available in a local file or on a server,

How do I pass an iterable to the DataFrame class?

Table 5.1 shows three functions that can be used to create a DataFrame from existing iterables, namely:

- `.DataFrame.from_dict`, which creates a DataFrame from a dictionary
- `.DataFrame.from_records`, which creates a DataFrame from a list of tuples
- `.DataFrame.sparse.from_spmatrix`, which creates a DataFrame from a SciPy sparse matrix (a convenient option when it comes to manipulate network data, which, oftentimes, exhibit sharp sparseness)

TABLE 5.1
Creating a DataFrame from Data Loaded in the Python Session

Routine	Synopsis
`pd.DataFrame.from_dict`	Construct DataFrame from dict of array-like or dicts
`pd.DataFrame.from_records`	Convert structured or record ndarray to DataFrame
`pd.DataFrame.sparse.from_spmatrix`	Create a new DataFrame from a SciPy sparse matrix (helpful for network data)

Notes: the statements included in the 'Routine' column assume Pandas is loaded with the `pd` alias.

Can you show me `.DataFrame.from_dict` in action?

Snippet 6.1 shows how to create a DataFrame from one or more iterables loaded in the current Python session. In line 5, we create a dictionary whose keys are associated with variables — i.e., the columns of the DataFrame. In line 10, we assign the variable `df` to the output of `.from_dict` with the dictionary `my_data` as input. In line 11, `df` is printed to the screen. You may have noticed that the mathematical progression reported on the left-right of the tabular data is the so-called Pandas 'index,' a concept we will see in Section 5.3.

By default, `.from_dict` parses the values of the input dictionary as columns. However, it is also possible to parse a dictionary's values as cases. We achieve that in line 30, where we populate the discretionary parameter `orient` with the value `index`, meaning Python must consider the dictionary keys as cases rather than columns. In line 38, we make a further adjustment by passing a list with the column names to the discretionary parameter `columns`. It is not necessary to do that, but it helps to interpret the columns — which, when we set `orient="index"`, are named with a mathematical progression by default.

Snippet 6.1 — creating a DataFrame from a dictionary

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # create a dictionary of arrays
5  >>> my_data = {
6          "var_1": [1, 2, 3, 4, 5],
7          "var_2": ["ABC", "Hello world", "Bazinga!", "cheers", "ciao"],
8      }
9  # get a DataFrame from the dictionary and display it
10 >>> df = pd.DataFrame.from_dict(my_data)
11 >>> df
12    var_1        var_2
13 0      1          ABC
14 1      2  Hello world
15 2      3      Bazinga
16 3      4       cheers
17 4      5         ciao
18
19 # create a dictionary whose keys are cases
20 >>> my_data = {"case_1": ["Pluto", "dog"], "case_2": ["Goofy", "dog"]}
21
22
23 # get a DataFrame from the dictionary and display it
24 >>> df = pd.DataFrame.from_dict(my_data)
25 >>> df
26    case_1   case_2
```

```
0   Pluto    Goofy
1     dog      dog

# ... something wrong here - let us adjust the optional param 'orient'
>>> df = pd.DataFrame.from_dict(my_data, orient="index")
>>> df
            0       1
case_1  Pluto     dog
case_2  Goofy     dog

# ... still something wrong here - where are the column names?
>>> df = pd.DataFrame.from_dict(
        my_data, orient="index",
        columns=["name", "species"]
        )
>>> df
         name  species
case_1  Pluto      dog
case_2  Goofy      dog
```

How do I create a DataFrame from 'external' data?

As we will see in Section 5.13, there are plenty of Pandas IO utilities that target alternative file formats/extensions (e.g., `.json`, `.csv`, `.xlsx`). In this section, I focus on one specific case: creating a DataFrame from a CSV file. In the first part of the snippet, up until line 27, create fake data and write them to a local .csv file. Creating a DataFrame from a CSV file is a simple matter of passing the name of the file (or a file path) to the `.read_csv`. Such a function has a substantial number of discretionary parameters — I warmly encourage you to go through the documentation and familiarize with the various options.

Snippet 6.2 — creating a DataFrame from a CSV file

```
# impoort pandas with the socially accepted alias 'pd'
>>> import pandas as pd

# create fake data and write them to a .csv
# --+ column names
>>> columns = ["x", "y"]
# --+ column values
>>> x = [0, 1, 2]
>>> y = ["A", "B", "C"]
# --+ write the data to a file
>>> with open("my_data.csv", "w") as f:
        # write the column names first
        f.write(",".join(columns) + "\n")
        # then, write the data case-by-case
        for i, j in zip(x, y):
```

```
            f.write("{},{}".format(i, j) + "\n")
# --+ close the pipe
f.close()

# create a DataFrame from the .csv file and display it
>>> df = pd.read_csv("my_data.csv")
>>> df
   x  y
0  0  A
1  1  B
2  2  C
```

5.3 Checking a DataFrame's Attributes

What are the key attributes of a DataFrame?

Pandas DataFrame can be characterized along several attributes, such as:

- Shape
- Size
- Number of dimensions
- Set of column names
- Set of case indices

A convenient way to access multiple DataFrame's attributes in a row is by using the `.info` method. Per Snippet 6.3 (see lines 18-26), we know that `df` has three entries and as many indices, two columns (an object, `country`, and a float `gdp_pc`), and occupies circa 176 bytes in memory.

The bottom section of Snippet 6.3 shows how to access the individual attributes of a DataFrame using `.shape`, `.size`, `.ndim`, and `.memony_usage`.

Snippet 6.3 — accessing a DataFrame's attributes with `.info`

```
# import pandas with the socially accepted alias 'pd'
>>> import pandas as pd

# create a DataFrame from a dictionary
>>> gdp_data = {
        "country": ["Belgium", "France", "Germany"],
        "gdp_pc": [51767.8, 43518.5, 50801.8]
    }
>>> df = pd.DataFrame.from_dict(gdp_data)
>>> df
   country   gdp_pc
0  Belgium  51767.8
1   France  43518.5
```

```
2  Germany  50801.8

# get DataFrame 'info'
>>> df.info()
RangeIndex: 3 entries, 0 to 2
Data columns (total 2 columns):
     Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   country  3 non-null      object
 1   gdp_pc   3 non-null      float64
dtypes: float64(1), object(1)
memory usage: 176.0+ bytes

# let us get df's attributes one-by-one
# --+ shape (cases by columns)
>>> df.shape
(3, 2)
# --+ size (cases X columns)
>>> df.size
6
# --+ number of dimensions (columns)
>>> df.ndim
2
# --+ check memory usage column-by-column
>>> df.memory_usage()
Index      128
country     24
gdp_pc      24
dtype: int64
```

5.4 The Anatomy of a DataFrame

What are the components of a DataFrame?

Mainly, DataFrames have two components:

- An `index`
- A set of columns, each of which is a `Pandas Series`

What is the index of a DataFrame?

Put simple, it is a Python object associated with a case or cases in a DataFrame. The index is the primary key for a DataFrame.

How does a DataFrame index matter?

The index is the primary key for a DataFrame and makes data querying easier and more efficient (we will see this in Section 5.5.

Is it mandatory to pass an index when I create a DataFrame?

In general, it is not necessary. When we do not pass an index, Pandas will create a mathematical progression and assign it to the index (see Snippet 6.3).

How do I access a DataFrame index?

`index` is a DataFrame's attribute (in a Pythonic sense!). Snippet 6.4 shows how to access the index (line 13). Also, the snippet shows that a `RangeIndex` object is an iterable object (line 17).

Can I edit a DataFrame index?

Yes, you can. Snippet 6.4 illustrates assigning a DataFrame to an iterable object (line 24). To change the index, it is also possible to use the function `.set_index` (see Section 5.6.)

Snippet 6.4 — accessing and amending a DataFrame's index

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # create a DataFrame from a dictionary
5  >>> df = pd.DataFrame.from_dict({"S":["s1", "s2", "s3"], "X":[-99, 8, 0]})
6  >>> df
7      S   X
8  0  s1 -99
9  1  s2   8
10 2  s3   0
11
12 # access the index
13 >>> df.index
14 RangeIndex(start=0, stop=3, step=1)
15
16 # iterate over the index
17 >>> for item in df.index:
18 ...     print(item)
19 0
20 1
21 2
22
23 # change the index
24 >>> df.index = ["case_1", "case_2", "case_3"]
25 >>> df
26          S   X
27 case_1  s1 -99
28 case_2  s2   8
29 case_3  s3   0
```

What is a Pandas Series?

A `Pandas Series` is a one-dimensional object. The columns of a DataFrame are a collection of Series objects.

How do I create a Series?

It is possible to create a Series using the `.series` class (see Snippet 6.5, line 5).

How do I access a Series included in DataFrame?

A Series object can be accessed as a DataFrame attribute (see Snippet 6.5, line 13). It is self-evident that a Series borrows the index of its parent DataFrame (see lines 16 and 24).

Snippet 6.5 — accessing a Series included in a DataFrame

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # create a series
5  >>> S = pd.Series(['s1', 's2', 's3'])
6  >>> print(S)
7  0    s1
8  1    s2
9  2    s3
10
11 # accessing a DataFrame column as a Series
12 # --+ the data
13 >>> df = pd.DataFrame.from_dict({"S":["s1", "s2", "s3"], "X":[-99, 8, 0]})
14 # --+ assign S to the fetched column and print S
15 >>> S = df.S
16 >>> print(S)
17 0    s1
18 1    s2
19 2    s3
20 # --+ amend the index
21 >>> df.index = ["case_1", "case_2", "case_3"]
22 # --+ assign S to the fetched column and print S
23 >>> S = df.S
24 >>> print(S)
25 case_1    s1
26 case_2    s2
27 case_2    s3
```

5.5 Querying DataFrames

What is data querying?

Data queries can take two non-mutually exclusive forms. One may want to:

- Select a subset of cases
- Select a subset of columns

How do I query data in Pandas?

Pandas DataFrame objects have a property called `.loc` that allows accessing a group of rows and columns by label(s) or a boolean array. To use `.loc`, one has to pass two inputs among brackets:

- The set of cases to select (the left-hand side element among brackets)
- The set of columns to select (the right-hand side element among brackets)

Snippet 6.6 shows a couple of use cases for the `.loc` property:

- Line 33 selects the first two cases of the DataFrame
- Line 39 selects the column "price" first two rows of the DataFrame
- Line 46 selects the columns "price" and "color" for all cases included in the DataFrame
- Line 52 selects the cases for which the column "price" is less than or equal to 8.00
- Line 59 selects the cases for which the column "price" is less than 9.00 **and** the column "color" is equal to "green"
- Line 65 expands on line 59 by selecting the column "price" when "price" is less than 9.00 **and** "color" is equal to "green"

Please refer to sections 3.8 and 3.9 to consolidate your knowledge on the Python syntax of statements and control flow.

Snippet 6.6 — querying data in Pandas with `.loc`

```
1 # import pandas with the socially accepted alias 'pd'
2 >>> import pandas as pd
3
4 # create a DataFrame from a dictionary
5 >>> df = pd.DataFrame.from_dict(
```

```
        {
            "product": ["a", "b", "c"],
            "price": [9.87, 8.63, 6.45],
            "color": ["green", "green", "blue"],
        }
    )

# data preview
>>> df
  product  price  color
0       a   9.87  green
1       b   8.63  green
2       c   6.45   blue

# info
>>> df.info()
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
     Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   product  3 non-null      object
 1   price    3 non-null      float64
 2   color   3 non-null      object
dtypes: float64(1), object(2)
memory usage: 200.0+ bytes

# select the first the two cases using a range of indices
>>> df.loc[0:2]
      product  price  color
    0       a   9.87  green
    1       b   8.63  green

# select the column "price" for the first two cases using a boolean array
>>> df.loc[0:2, "price"]
0    9.87
1    8.63
2    6.45
Name: price, dtype: float64

# select the columns "price" and "color" for all cases
>>> df.loc[:, ["price", "color"]]
   price  color
0   9.87  green
1   8.63  green

# select all cases for which the column "price" is greater than or equal to 8.00
>>> df.loc[df["price"] >= 8.00, ]
  product  price  color
0       a   9.87  green
1       b   8.63  green

# select all cases for which the column "price" is less than 9.00 and
```

```
58 # "color" is equal to "green"
59 >>> df.loc[(df["price"] < 9.00) & (df["color"] == "green" ), ]
60   product  price  color
61 1       b   8.63  green
62
63 # select all cases for which the column "price" is less than to 9.00 and
64 # "color" is equal to "green"; also, keep the column "price" only
65 >>> df.loc[(df["price"] < 9.00) & (df["color"] == "green" ), "price"]
66 1    8.63
67 Name: price, dtype: float64
```

5.6 Manipulating DataFrame Columns

What are the main operations on a DataFrame's columns?

Mainly, one may want to:

- Rename a column
- Drop a column
- Create a new column
- Amend an existing column's values

How do I rename or drop a column in Pandas?

As shown in Snippet 6.7, Pandas has two methods, `.rename()` and `.drop()`, to rename and drop columns respectively.

Line 35 shows how to rename a column. A first argument is a dictionary mapping the new name onto the old name (the dictionary's key). The second argument is a boolean flag that indicates whether to make the change effective (if "True") or not (if "False").

Lines 45 and 51 show how to drop a column. The first argument is an array with the name(s) of the column(s) to delete. The second argument is a boolean flag that indicates whether to make the change effective (if "True") or not (if "False").

Snippet 6.7 — renaming and dropping DataFrame columns

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # the df
5  >>> df = pd.DataFrame(
6          {
7              "laptop": ["MacBook Pro 13inch", "Thinkpad T14", "Dell XPS 13"],
8              "ram": ["16 GB", "48 GB", "8 GB"],
9              "os": ["macOS Monterey", "Debian 11", "Windows 11"],
10             "chip": ["M1", "Ryzen", "Intel Core i7"]
11         }
```

```
    )

# data view
>>> df
                laptop    ram               os           chip
0  MacBook Pro 13inch  16 GB  macOS Monterey             M1
1         Thinkpad T14  48 GB       Debian 11          Ryzen
2          Dell XPS 13   8 GB      Windows 11  Intel Core i7

# info
>>> df.info()
RangeIndex: 3 entries, 0 to 2
Data columns (total 4 columns):
     Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   laptop  3 non-null      object
 1   ram     3 non-null      object
 2   os      3 non-null      object
 3   chip    3 non-null      object
dtypes: object(4)
memory usage: 224.0+ bytes

# rename the column "ram" to "memory"
>>> df.rename(columns={"ram": "memory"}, inplace=True)
>>> df
                laptop memory               os           chip
0  MacBook Pro 13inch  16 GB  macOS Monterey             M1
1         Thinkpad T14  48 GB       Debian 11          Ryzen
2          Dell XPS 13   8 GB      Windows 11  Intel Core i7

# drop the column "chip"
# --+ when inplace is set to False, the outcom of .drop is sent to the
# interactive sessions, but the data in memory are not affected
>>> df.drop(columns=["chip"], inplace=False)
                laptop memory               os
0  MacBook Pro 13inch  16 GB  macOS Monterey
1         Thinkpad T14  48 GB       Debian 11
2          Dell XPS 13   8 GB      Windows 11
# --+ when inplace is set to True, the data in memory are affected
>>> df.drop(columns=["chip"], inplace=True)
>>> df
                laptop memory               os
0  MacBook Pro 13inch  16 GB  macOS Monterey
1         Thinkpad T14  48 GB       Debian 11
2          Dell XPS 13   8 GB      Windows 11
```

How do I create a new column or amend an existing Pandas column?

The `.loc` method is the most appropriate way to create a new column or alter an existing column's values. Snippet 6.8 illustrates a couple of common tasks. In line 15, we assign a list of string objects to a new column called `manufacturer` for all cases included in `df`.

In lines 24/26, 41, and 58 we amend the value of an existing column. First, we manipulate the strings included under RAM to replace the substring `"\sGB"` with an empty string (lines 24/26). Second, we change the type of the column from `string` to `integer` using the Pandas' `.astype()` method (line 41). Then, we take the log of `ram` and assign it to the new column `log_ram` (line 58).

In line 85, we use `.loc` to create a new variable for a subset of cases only. Specifically, we populate the column `gpu` provided the value of `laptop` is "Thinkpad T14."

Snippet 6.8 — creating and amending DataFrame columns

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # the df
5  >>> df = pd.DataFrame(
6          {
7              "laptop": ["MacBook Pro 13inch", "Thinkpad T14", "Dell XPS 13"],
8              "ram": ["16 GB", "48 GB", "8 GB"],
9              "os": ["macOS Monterey", "Debian 11", "Windows 11"],
10             "chip": ["M1", "Ryzen", "Intel Core i7"],
11         }
12 )
13
14 # create a new column, e.g., the name of the manufacturer
15 >>> df.loc[:, "manufacturer"] = ["Apple", "Lenovo", "Dell"]
16 >>> df
17                laptop ram              os           chip manufacturer
18 0  MacBook Pro 13inch  16  macOS Monterey             M1        Apple
19 1        Thinkpad T14  48       Debian 11          Ryzen       Lenovo
20 2         Dell XPS 13   8     Windows 11  Intel Core i7         Dell
21
22 # transform the column "ram" from string to number type
23 # --+ step 1: get rid of non-number characters using a regular expression
24 >>> df.loc[:, "ram"] = df.loc[:, "ram"].str.replace(r"[^0-9]", "")
25 # --+ alternative way to carry out step 1: using Pandas string methods
26 >>> df.loc[:, "ram"] = df["ram"].str.replace(r"\sGB", "")
27 # --+ step 2: convert the string to number type
28 # ----+ check the type of "ram"
29 >>> df.info()
30 Data columns (total 5 columns):
31      Column        Non-Null Count  Dtype
```

```
---  ------        --------------  -----
 0   laptop        3 non-null      object
 1   ram           3 non-null      object
 2   os            3 non-null      object
 3   chip          3 non-null      object
 4   manufacturer  3 non-null      object
dtypes: object(5)
# ----+ surprise-surprise: we have not a number type yet
# let us change dtype with Pandas astype method
>>> df.loc[:, "ram"] = df.loc[:, "ram"].astype(int)
# ----+ check the data type again
>>> df.info()
Data columns (total 5 columns):
     Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   laptop        3 non-null      object
 1   ram           3 non-null      int64
 2   os            3 non-null      object
 3   chip          3 non-null      object
 4   manufacturer  3 non-null      object
dtypes: int64(1), object(4)
memory usage: 248.0+ bytes
memory usage: 248.0+ bytes

# transform an existing column and assign the output to a new column
# --+ import numpy to access the log function
>>> df["log_ram"] = np.log(df["ram"])
# --+ preview
>>> df.info()
               laptop  ram              os           chip manufacturer  \
0  MacBook Pro 13inch   16  macOS Monterey             M1        Apple
1        Thinkpad T14   48       Debian 11          Ryzen       Lenovo
2         Dell XPS 13    8     Windows 11  Intel Core i7         Dell

    log_ram
0  2.772589
1  3.871201
2  2.079442
# --+ info
>>> df.info()
Data columns (total 6 columns):
     Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   laptop        3 non-null      object
 1   ram           3 non-null      int64
 2   os            3 non-null      object
 3   chip          3 non-null      object
 4   manufacturer  3 non-null      object
 5   log_ram       3 non-null      float64
dtypes: float64(1), int64(1), object(4)
memory usage: 272.0+ bytes

```

```
# create a new column conditional on another column's value
>>> df.loc[df["laptop"] == "Thinkpad T14", "gpu"] = True
# --+ info
>>> df.info()
Data columns (total 7 columns):
     Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   laptop        3 non-null      object
 1   ram           3 non-null      int64
 2   os            3 non-null      object
 3   chip          3 non-null      object
 4   manufacturer  3 non-null      object
 5   log_ram       3 non-null      float64
 6   gpu           1 non-null      object
dtypes: float64(1), int64(1), object(5)
memory usage: 296.0+ bytes
# --+ preview
>>> df
                laptop  ram              os           chip manufacturer  \
0  MacBook Pro 13inch   16  macOS Monterey            M1        Apple
1        Thinkpad T14   48       Debian 11          Ryzen       Lenovo
2         Dell XPS 13    8     Windows 11  Intel Core i7         Dell

    log_ram   gpu
0  2.772589   NaN
1  3.871201  True
2  2.079442   NaN
```

What is a missing value?

A missing value is a datapoint for which no information is available in the dataset. Missing values can arise for many reasons. The most popular reasons are:

- the case does not present a value for the variable. For example, it is not possible to record the stock market of a company before the IPO
- the case does present a value for the variable. However, the value was not recorded because of the limitations of the data gathering process
- the recorded value is not accurate/valid — hence, it was removed from the dataset

How does Pandas represent missing values?

Pandas denote missing values using NumPy's floating representation `numpy.NaN`

How do I handle missing values in Pandas?

There are several approaches to cope with missing values:

- deleting all cases where a missing value is present
- replacing missing values with a fixed value
- replacing missing values with the mean of the column
- replacing missing values with the estimate of a statical model

The latter approach is the most sophisticated one and falls beyond the remit of these notes. In Snippet 6.9, I show how to implement the first three approaches in Pandas. Line 34 uses the `.dropna()` method to delete all cases presenting at least one missing value. Line 41 draws on the `.fillna()` method to replace the missing values with a fixed value (e.g., a value that the analyst considers 'plausible' based on her/his contextual knowledge). Like line 41, line 50 relies the `.fillna()` method. However, it does not replace the missing values with a fixed value. Instead, it uses the mean value of `price`.

Snippet 6.9 — handling missing values

```
1  .
2  # import numpy with the socially accepted alias "np"
3  >>> import numpy as np
4  # import pandas with the socially acceptable alias "pd"
5  >>> import pandas as pd
6
7  # the dataframe
8  >>> df = pd.DataFrame(
9          {
10             "item": ["a", "b", "c", "d", "e"],
11             "price": [16.32, 16.78, np.nan, np.nan, 16.41],
12         }
13 )
14
15 # data view
16 >>> df
17   item  price
18 0    a  16.32
19 1    b  16.78
20 2    c    NaN
21 3    d    NaN
22 4    e  16.41
23
24 # info
25 >>> df.info()
```

```
Data columns (total 2 columns):
     Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   item    5 non-null      object
 1   price   3 non-null      float64
dtypes: float64(1), object(1)
memory usage: 208.0+ bytes

# approach 1 --- delete all cases where a missing value is present
>>> df.dropna()
  item  price
0    a  16.32
1    b  16.78
4    e  16.41

# approach 2 --- replace missing values with a fixed value
>>> df.fillna(value=16.99, inplace=False)
  item  price
0    a  16.32
1    b  16.78
2    c  16.99
3    d  16.99
4    e  16.41

# approach 3 --- replace missing values with the mean of the columns
>>> df.fillna(value=np.mean(df["price"]), inplace=False)
  item      price
0    a  16.320000
1    b  16.780000
2    c  16.503333
3    d  16.503333
4    e  16.410000
```

5.7 Data Types and Pandas

What are the data types admitted in a Pandas object?

Mainly, Pandas uses NumPy arrays as the concrete objects contained with an Index, Series, or DataFrame. To recall the various NumPy dtypes, please refer to Sections 4.2.

Can a Series contain multiple data types?

Yes, a Series can contain multiple data types (hence, a DataFrame can). In this sense, Pandas offers a more flexible data structure than NumPy arrays, which must contain objects of the same type. Snippet 6.10 shows that a list with mixed types is downgraded to a NumPy array with string objects. Instead, a Series preserves the type of each object (see line 22).

Snippet 6.10 — multiple data types in a Series

```
# import numpy and pandas with the socially accepted alias 'np' and 'pd'
>>> import numpy as np
>>> import pandas as pd

# a list with a string, a float, and an integer
>>> L = ["xyz", -17.64, 0]

# create a NumPy array from the list
>>> A = np.array(L)
>>> print(A)
['xyz' '-17.64' '0']

# create a Series from the list
>>> S = pd.Series(L)
>>> print(S)
0      xyz
1   -17.64
2        0

# proof that Series preserves the type of the individual object
# --+ fetch the second item of the list and carry out a mathematical operation
>>> S[1]/2
-8.82
```

5.8 Handling 'Time' Data in Pandas

How good is Pandas at handling 'time' data?

How many types of 'time' data are supported in Pandas?

Pandas excels at handling time data, a key component of time series and panels.

Pandas has three kinds of objects that can be used to store and represent information on time-related quantities:

- .Timestamp, the Pandas replacement for Python's `Datatime` object, represent a specific point in time
- `.Timedelta` represents two dates or time
- `.Period` represents a period of time

Can you give me an example of `.Timestamp` and `.Timedelta` objects?

Snippet 6.11 illustrates how to initialize and use `.Timestamp` and `.Timedelta` objects. The output of line 30 shows that the column `timestamp` has been parsed as a string. At this point, we cannot perform any operation on it (e.g., getting the time elapsed since an event). Hence, in line 43 we pass the values under the column `timestamp` to `.Timestamp`. To apply the transformation column-wise, we draw on the `.apply()` method, which takes a function as an argument. Specifically, we build our function using a `lambda expression` getting the `.Timestamp` of an element `s` (that is, the individual values included in `timestamp`). The output of line 45 indicates that, after the transformation, the column `timestamp` contains time quantities.

The remainder of the snippet illustrates how to extract the individual attributes of a `.Timestamp` object (see lines 58 - 68) and create a `.Timedelta` object (see line 88).

Snippet 6.11 — working with Timestamp and Timedelta objects

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # fake product review data
5  >>> df = pd.DataFrame.from\_dict(
6          {
7              "timestamp": [
8                  "2017-10-08 12:03:05",
9                  "2020-09-07 08:09:45",
10                 "2021-04-11 10:12:13",
11             ],
12             "product": [
13                 "Darth Vader plush",
14                 "Obi-Wan Kenobi lightsaber",
15                 "Yoda pijamas"
16         ],
17             "reviewer": ["Sheldon", "Sheldon", "Leonard"],
18             "rating": [1, 2, 5],
19         }
20     )
21
22 # data view
23 >>> df.head()
24              timestamp                    product reviewer  rating
25 0  2017-10-08 12:03:05          Darth Vader plush  Sheldon       1
26 1  2020-09-07 08:09:45  Obi-Wan Kenobi lightsaber  Sheldon       2
27 2  2021-04-11 10:12:13               Yoda pajamas  Leonard       5
28
29 # get basic info
30 >>> df.info()
```

```
RangeIndex: 3 entries, 0 to 2
Data columns (total 4 columns):
     Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   timestamp  3 non-null      object
 1   product    3 non-null      object
 2   reviewer   3 non-null      object
 3   rating     3 non-null      int64
dtypes: int64(1), object(3)
memory usage: 224.0+ bytes

# convert 'timestamp' from string to Timestamp type
>>> df.loc[:, "timestamp"] = df["timestamp"].apply(lambda s: pd.Timestamp(s))
# info shows that timestamp has datetime dtype
>>> df.info()
Data columns (total 4 columns):
     Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   timestamp  3 non-null      datetime64[ns]
 1   product    3 non-null      object
 2   reviewer   3 non-null      object
 3   rating     3 non-null      int64
dtypes: datetime64[ns](1), int64(1), object(2)
memory usage: 224.0+ bytes

# extract the timestamp components and assign them to new columns
# --+ year
>>> df.loc[:, "year"] = df["timestamp"].dt.year
# --+ month
>>> df.loc[:, "month"] = df["timestamp"].dt.month
# --+ day
>>> df.loc[:, "day"] = df["timestamp"].dt.day
# --+ hour
>>> df.loc[:, "hour"] = df["timestamp"].dt.hour
# --+ hour
>>> df.loc[:, "minute"] = df["timestamp"].dt.minute
# --+ second
>>> df.loc[:, "second"] = df["timestamp"].dt.second
# --+ data view
            timestamp                    product reviewer  rating  year  \
0 2017-10-08 12:03:05          Darth Vader plush  Sheldon       1  2017
1 2020-09-07 08:09:45  Obi-Wan Kenobi lightsaber  Sheldon       2  2020
2 2021-04-11 10:12:13               Yoda pijamas  Leonard       5  2021

   month  day  hour  minute  second
0     10    8    12       3       5
1      9    7     8       9      45
2      4   11    10      12      13

# calculate the time elapsed since the product launch and the review
# --+ fake product launch timestamps
>>> df.loc[:, "launch"] = pd.to_datetime(
```

```
        ["2011-11-01 08:45:19", "2012-02-07 13:07:07", "2011-05-10 13:04:05"]
        )
# --+ get launch as a Timestamp object
>>> df.loc[:, "launch"] = df["launch"].apply(lambda s: pd.Timestamp(s))
# --+ here is a Timedelta object
>>> df.loc[:, "deltat"] = df["timestamp"] - df["launch"]
# --+ data view
            timestamp                    product reviewer  rating  year  \
0 2017-10-08 12:03:05          Darth Vader plush  Sheldon       1  2017
1 2020-09-07 08:09:45  Obi-Wan Kenobi lightsaber  Sheldon       2  2020
2 2021-04-11 10:12:13               Yoda pijamas  Leonard       5  2021

   month  day  hour  minute  second              launch             deltat
0     10    8    12       3       5 2011-11-01 08:45:19 2168 days 03:17:46
1      9    7     8       9      45 2012-02-07 13:07:07 3134 days 19:02:38
2      4   11    10      12      13 2011-05-10 13:04:05 3623 days 21:08:08
# --+ data info
>>> df.info()
Data columns (total 12 columns):
     Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   timestamp  3 non-null      datetime64[ns]
 1   product    3 non-null      object
 2   reviewer   3 non-null      object
 3   rating     3 non-null      int64
 4   year       3 non-null      int64
 5   month      3 non-null      int64
 6   day        3 non-null      int64
 7   hour       3 non-null      int64
 8   minute     3 non-null      int64
 9   second     3 non-null      int64
 10  launch     3 non-null      datetime64[ns]
 11  deltat     3 non-null      timedelta64[ns]
dtypes: datetime64[ns](2), int64(7), object(2), timedelta64[ns](1)
memory usage: 416.0+ bytes
```

Can you give me an example of a `.Period` object?

`.Period` takes a string as input and returns an object of class `Period`, which has many attributes to establish the unit of time a timestamp falls on. For example, it is possible to assess a period's associated 'day' (see Snippet 6.12, line 12), 'day of the week' (line 16), 'month' (line 20), 'quarter' (line 24), or 'week of the year' (line 28).

Snippet 6.12 — working with Period objects

```
1  # import pandas with the socially acceptable alias pd
2  >>> import pandas as pd
3
4  # a timestamp as a string
5  >>> s = "2011-11-01 23:17:01"
6
7  # get a period object
8  >>> p = pd.Period(s, freq="s")
9
10 # extract sample information from p
11 # --+ calendar day
12 >>> p.day
13 1
14
15 # --+ day of the week (Monday is 0)
16 >>> p.dayofweek
17 1
18
19 # --+ month
20 >>> p.month
21 11
22
23 # --+ quarter
24 >>> p.quarter
25 4
26
27 # --+ week of the year (ranging from 0 to 59)
28 >>> p.weekofyear
29 4
```

5.9 Shaping and Reshaping DataFrames

What is 'panel data'?

Often, we work with panel data, wherein the same statistical unit (e.g., an employer, a company, an administrative city) is observed multiple times. An example is a security's price taken at different times of the day.

What are the alternative ways for representing panel data?

Mainly, there are two approaches to representing a panel dataset:

- The *wide* data structure consists of creating one row per unit and as many columns as measurement occasions. Figure 5.2 provides a pictorial representation of the wide data structure
- The *long* data structure consists of creating pairs of measurement occasions and values nested within the same unit. In other words, the same unit is repeated across as many rows as measurement occasions. Figure 5.3 provides a pictorial representation of the long data structure

Case	p_0	p_1	$\cdots$	p_i	$\cdots$	p_k
1	$p_{1,0}$	$p_{1,1}$	$\cdots$	$p_{1,i}$	$\cdots$	$p_{1,k}$
2	$p_{2,0}$	$p_{2,1}$	$\cdots$	$p_{2,i}$	$\cdots$	$p_{2,k}$
j	$p_{j,0}$	$p_{j,1}$	$\cdots$	$p_{j,i}$	$\cdots$	$p_{j,k}$
n	$p_{n,1}$	$p_{n,2}$	$\cdots$	$p_{n,j}$	$\cdots$	$p_{n,k}$

Figure 5.2: A stylized representation of a 'wide' data structure

What are the pros and cons of a wide data structure?

- Pros:
  - easier to work with for people coming from a spreadsheet environment
  - going through a row, one can make sense of the trend of a particular unit
- Cons:
  - the larger the number of measurement of occasions, the less readable the file, the larger the number of columns ($\rightarrow$ the more difficult to work with the data)
  - data manipulation activities must carry out column-by-column

What are the pros and cons of a wide data structure?

- Pros:
  - data manipulation activities are easier to carry out since there is only one column per case
  - it is easier to group and aggregate the data around cases
- Cons:
  - it is harder to inspect visually the data
  - this data structure is less familiar to people without information systems or programming background

Case	Time	Value
1	0	$p_{1,0}$
1	1	$p_{1,1}$
1	2	$p_{1,2}$
...	...	...
1	i	$p_{1,i}$
...	...	...
1	k	$p_{1,k}$
2	0	$p_{2,0}$
2	1	$p_{2,1}$
2	2	$p_{2,2}$
...	...	...
2	i	$p_{2,i}$
...	...	...
2	k	$p_{2,k}$
j	0	$p_{j,0}$
j	1	$p_{j,1}$
j	2	$p_{j,2}$
...	...	...
j	i	$p_{j,i}$
...	...	...
j	k	$p_{j,k}$
n	0	$p_{n,0}$
n	1	$p_{n,1}$
n	2	$p_{n,2}$
...	...	...
n	i	$p_{n,i}$
...	...	...
n	k	$p_{n,k}$

Figure 5.3: A stylized representation of a 'long' data structure

How can I move from a wide to a long data structure or the other way around?

To get a long data structure, one may want to use the Pandas function `pd.melt()`, and, to achieve a wide structure, the Pandas function `pd.pivot()`.

Per Snippet 6.13, typically we pass four inputs to `pd.melt()` (see line 16):

- `frame`: the source DataFrame to re-arrange
- `id_vars`: the column(s) that will be used to identify the cases
- `var_name`: the name of the target DataFrame column that will contain the information regarding the measurement occasions (i.e., the time)
- `value_name`: the name of the source DataFrame column containing the values to re-arrange

Snippet 6.13 also passes from a long to a wide data structure. In line 47, we pass the following inputs to `pd.pivot()`:

- `data`: the source DataFrame to re-arrange
- `index`: the column(s) that will be used to identify the cases
- `columns`: the name of the source DataFrame column(s) whose categories will be used to create the columns of the target DataFrame
- `values`: the name of the source DataFrame column containing the values to re-arrange

Snippet 6.13 — from wide data to long and back

```
1  # import pandas with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # create a wide data frame
5  >>> df = pd.DataFrame.from_dict(
6          {
7              "product": ["a", "b", "c"],
8              "t0sales": [10, 5, 0],
9              "t1sales": [7, 6, 5],
10             "t2sales": [4, 6, 9],
11         }
12     )
13
14 # get a long data frame
15 # --+ let us use the melt method
16 >>> molten = pd.melt(
```

```
        frame = df, id_vars=["product"], var_name="time", value_name="sales"
    )
# --+ the result is a long data frame
>>> molten
  product     time  sales
0       a  t0sales     10
1       b  t0sales      5
2       c  t0sales      0
3       a  t1sales      7
4       b  t1sales      6
5       c  t1sales      5
6       a  t2sales      4
7       b  t2sales      6
8       c  t2sales      9
# --+ the time column is not clean, though. Let us clean it
# ----+ get rid of non-numeric characters
>>> molten.loc[:, "time"] = molten["time"].str.replace("t|sales", "")
# ----+ convert the time column to an integer
>>> molten.loc[:, "time"] = molten["time"].astype(int)
# ----+ get info
Data columns (total 3 columns):
     Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   product  9 non-null      object
 1   time     9 non-null      int64
 2   sales    9 non-null      int64
dtypes: int64(2), object(1)
memory usage: 344.0+ bytes
# ----+ data view
>>> molten
  product  time  sales
0       a     0     10
1       b     0      5
2       c     0      0
3       a     1      7
4       b     1      6
5       c     1      5
6       a     2      4
7       b     2      6
8       c     2      9

# from long data to wide
>>> wide = pd.pivot_table(
        data = molten, index="product", columns="time", values="sales"
    )
time      0  1  2
product
a        10  7  4
b         5  6  6
c         0  5  9
```

5.10 Group By Part I: Data Aggregation

Long data re-crossing the road?

As we saw in the previous section (5.9), long data present a 'nested structure' since values cluster around cases.

So what?

The nested structure of the data plays a key role in aggregating the data. For example, one may want to aggregate daily sales data into monthly or quarterly sales data. In so doing, one trades off the granularity of the data (e.g., daily sales records) for the possibility to best appreciate a general trend in the data (e.g., comparing the sales of two contiguous quarters).

Got it. How can I do this in Pandas?

Pandas has a class to help with this task, namely, `.groupby`. Mainly, this class captures the nested structure of the data and allows to aggregate or transform the data (see next section 5.11) according to the nested structure.

Snippet 6.14 shows how to use `.groupby` for aggregation purposes. The sample DataFrame contains monthly sales data for three products, each observed for three months (line 14). First, we must tell Pandas what grouping data to consider. We do that in line 27, wherein we pass the column name used to group the data. In this example, we use the `product` column. That means we will pass from product-time data points (i.e., monthly data) to product data points (e.g., the total sales for each product over the three months). Since the grouping structure is defined by one column only, the input is a string with the column's name. An array of strings must be passed if the grouping structure is based on multiple columns.

As shown in line 27, calling the object `gr` does not affect the data. Instead, the signature of the object is displayed. To aggregate the data, we must deploy `gr` as shown in line 33. The string between brackets is the target column whose values we want to consider; `.aggregate()` defines the operation to carry out based on the grouping; the array of NumPy functions is the set of mathematical operations to perform as the data are aggregated. In our example, we ask for the total sales for each product and the average monthly sales over the three months (see line 34).

Snippet 6.14 — data aggregation with groupby

```
# import pandas with the socially accepted alias 'pd'
>>> import pandas as pd

# create a long data frame
>>> df = pd.DataFrame.from_dict(
        {
            "product": ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c'],
            "month": [0, 0, 0, 1, 1, 1, 2, 2, 2],
            "sales": [10,  5,  0,  7,  6,  5,  4,  6,  9]
        }
    )

# data view
>>> df
  product  time  sales
0       a     0     10
1       b     0      5
2       c     0      0
3       a     1      7
4       b     1      6
5       c     1      5
6       a     2      4
7       b     2      6
8       c     2      9

# group by
>>> gr = df.groupby("product")
# --+ what do we get if we call gr?
>>> gr
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fb909d4f9d0>

# aggregate sales data around the grouping structure
>>> gr["sales"].aggregate([np.sum, np.mean])
         sum      mean
product
a         21  7.000000
b         17  5.666667
c         14  4.666667
```

5.11 Group By Part II: Data Transformation

Aggregate Vs. transform

Once we have created a grouping structure, we can use it for aggregating the data (i.e., passing from monthly to yearly data points, see the previous section 5.10) as well as for expanding the source data. This procedure is called transformation and does not affect the number of rows in the DataFrame.

In Snippet 6.15, we get the cumulative sum of a product's sales for each period (the output of line 28 shows the structure of the data). Having defined the grouping structure (line 41), we use the `.transform()` method to apply NumPy's `cumsum` function to the values of `sales` that regard the same product. As shown in line 44, the outcome of the transformation is assigned to the new column `cumulative_size`.

Snippet 6.15 — data transformation with groupby

```
1  # import the pandas library with the socially accepted alias 'pd'
2  >>> import pandas as pd
3
4  # the data frame
5  >>> df = pd.DataFrame.from_dict(
6          {
7              "product": ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c'],
8              "month": [0, 0, 0, 1, 1, 1, 2, 2, 2],
9              "sales": [10,  5,  0,  7,  6,  5,  4,  6,  9]
10         }
11     )
12
13 # data view
14 >>> df
15   product  month  sales
16 0       a      0     10
17 1       b      0      5
18 2       c      0      0
19 3       a      1      7
20 4       b      1      6
21 5       c      1      5
22 6       a      2      4
23 7       b      2      6
24 8       c      2      9
25
26
27 # sort the data by product and month values
28 >>> df.sort_values(by=["product", "month"], inplace=True)
29   product  month  sales
30 0       a      0     10
31 3       a      1      7
32 6       a      2      4
```

```
1       b      0      5
4       b      1      6
7       b      2      6
2       c      0      0
5       c      1      5
8       c      2      9

# group by
>>> gr = df.groupby("product")

# tranform the data
>>> df.loc[:, "cumulative_sales"] = gr["sales"].transform(np.cumsum)
>>> df
  product  month  sales  cumulative_sales
0       a      0     10                10
3       a      1      7                17
6       a      2      4                21
1       b      0      5                 5
4       b      1      6                11
7       b      2      6                17
2       c      0      0                 0
5       c      1      5                 5
8       c      2      9                14
```

5.12 Working with Multiple DataFrames

My data cut across data tables: how can I combine multiple them into a single DataFrame?

Often, the information we want to analyze is spread across different data tables. In this case, we must combine multiple DataFrame objects. Mainly, there are two approaches to data combination:

- Concatenate consists of stacking two or more data frames (either on the first axis or on the second axis)
- Merge consists of joining two or more data frames on a common set of keys

How do I concatenate two DataFrame objects in Pandas?

The `.concat()` method is used to concatenate two or more data frames. The method takes an array of DataFrames as input and returns a new data frame with the rows or columns from all the input data frames.

In Snippet 6.15, we carry out the following:

- We concatenate two DataFrame objects along the first axis (column-wise)
- We concatenate two DataFrame objects along the second axis (row-wise

concatenate two data frames. The first data frame contains sales data for the year's first quarter; the second data frame includes the information for the second quarter.

Snippet 6.16 — concatenating two DataFrame objects

```
# import pandas with the socially accepted alias pd
>>> import pandas as pd

# concatenate along the first axis (axis = 0)
# --+ the data frames
>>> df0 = pd.DataFrame.from_dict(
        {"quarter": ["first", "second"], "sales": [10, 30]}
    )
>>> df1 = pd.DataFrame.from_dict(
        {"quarter": ["third", "fourth"], "sales": [10, 40]}
    )
# --+ set the indices to match the data frames
>>> df0.set_index("quarter", inplace=True)
>>> df1.set_index("quarter", inplace=True)
# --+ concatenate the data frames
>>> pd.concat([df0, df1], axis=0)
         sales
quarter
first       10
second      30
third       10
fourth      40

# concatenate along the second axis (axis = 1)
# --+ the data frames
>>> df0 = pd.DataFrame.from_dict(
        {"product": [1, 2, 3], "sales": [10, 30, 20]}
    )
>>> df1 = pd.DataFrame.from_dict(
        {"product": [1, 2, 3], "units_sold": [3, 6, 2]}
    )
# --+ set the indices to match the data frames
```

```
>>> df0.set_index("product", inplace=True)
>>> df1.set_index("product", inplace=True)
# --+ the concatenated data frame
>>> pd.concat([df0, df1], axis=1)
         sales  units_sold
product
1           10           3
2           30           6
3           20           2
```

How do I merge two DataFrame objects in Pandas?

The `.merge()` method is used to merge two or more data frames. The method takes the following inputs:

- `left` and `right` are the DataFrae objects to be merged
- `on` is the column/list of columns on which the merge is to be performed
- `how` is the type of merge to be performed
  - `how="left"` — all cases from the left DataFrame are preserved irrespective of the fact they are present in the right DataFrame
  - `how="right"` — all cases from the right DataFrame are preserved irrespective of the fact they are present in the left DataFrame
  - `how = "inner"` — only the cases that are present in both the left and right DataFrame are preserved
  - `how = "outer"` — all cases from the left and right DataFrame are preserved irrespective of the fact they are present in the left or right DataFrame
- Optionally, one may want to set the parameters `left_index` and `right_index` to `True` in order to perform the merge on the indices of the DataFrames (default is `False`.)

Shaping and Reshaping DataFrames

```
# import pandas with the socially accepted alias pd
>>> import pandas as pd

# the data frames
>>> df0 = pd.DataFrame.from_dict(
        {"product": [1, 2, 3], "sales": [10, 30, 20]}
    )
>>> df1 = pd.DataFrame.from_dict(
        {"product": [2, 3, 4], "margin": [0.13, 0.31, 0.21]}
    )

# merge the data frames
# --+ keep all the cases from the left data frame
>>> pd.merge(df0, df1, on="propduct", how="left")
   product  sales  margin
0        1     10     NaN
1        2     30    0.13
2        3     20    0.31
# --+ keep all the cases from the right data frame
>>> pd.merge(df0, df1, on="propduct", how="right")
   product  sales  margin
0        2   30.0    0.13
1        3   20.0    0.31
2        4    NaN    0.21
# --+ keep all cases
>>> pd.merge(df0, df1, on="propduct", how="outer")
   product  sales  margin
0        1   10.0     NaN
1        2   30.0    0.13
2        3   20.0    0.31
3        4    NaN    0.21
# --+ keep the cases that are present in both data frames
>>> pd.merge(df0, df1, on="propduct", how="inner")
   product  sales  margin
0        2     30    0.13
1        3     20    0.31
# ---+ merge using the indices
# ----+ set the indices
>>> df0.set_index("product", inplace=True)
>>> df1.set_index("product", inplace=True)
# ----+ merge the data frames
>>> pd.merge(df0, df1, left_index=True, right_index=True)
         sales  margin
product
2           30    0.13
3           20    0.31
```

5.13 File Input and Output (IO) with Pandas

What type of data formats can I read/write with Pandas

There are plenty of options, including proprietary statistical languages such as SAS, SPSS, and Stata, Python's pickles, tabular data in CSV or TSV format, and data formats widely adopted online data sources, such as HTML, XML, and JSON. Table 5.2, 5.3, 5.4, 5.5 illustrate some of the most popular IO utilities.[28]

TABLE 5.2
Routines for Reading and Writing Data with Pandas: Pickles

Routine	Synopsis
`pd.read_pickle(filepath_or_buffer[, ...])`	Load pickled pandas object (or any object) from file
`df.to_pickle(path[, compression, ...])`	Pickle (serialize) object to file

Notes: the statements included in the 'Routine' column assume Pandas is loaded with the `pd` alias and there is a DataFrame loaded with name `df`.

TABLE 5.3
Routines for Reading and Writing Data with Pandas: Excel Spreadsheets

Routine	Synopsis
`pd.read_excel(io[, sheet_name, header, names, ...])`	
Read an Excel file into a pandas DataFrame. `df.to_excel(excel_writer[, ...])`	Write object to an Excel sheet
`pd.ExcelFile.parse([sheet_name, header, names, ...])`	Parse specified sheet(s) into a DataFrame
`df.Styler.to_excel(excel_writer[, sheet_name, ...])`	Write Styler to an Excel sheet
`df.ExcelWriter(path[, engine, date_format, ...])`	Class for writing DataFrame objects into excel sheets

Notes: the statements included in the 'Routine' column assume Pandas is loaded with the `pd` alias and there is a DataFrame loaded with name `df`.

TABLE 5.4
Routines for Reading and Writing Data with Pandas: JSON Files

Routine	Synopsis
`pd.read_json([path_or_buf, orient, typ, dtype, ...])`	Convert a JSON string to pandas object
`pd.json_normalize(data[, record_path, meta, ...])`	Normalize semi-structured JSON data into a flat table
`df.to_json([path_or_buf, orient, ...])`	Convert the object to a JSON string
`df.build_table_schema(data[, index, ...])`	Create a Table schema from data

Notes: the statements included in the 'Routine' column assume Pandas is loaded with the `pd` alias and there is a DataFrame loaded with name `df`.

TABLE 5.5
Routines for Reading and Writing Data with Pandas: Flat Files

Routine	Synopsis
`pd.read_table(filepath_or_buffer[, sep, ...])`	Read general delimited file into DataFrame
`pd.read_csv(filepath_or_buffer[, sep, ...])`	Read a comma-separated values (csv) file into DataFrame
`df.to_csv([path_or_buf, sep, na_rep, ...])`	Write object to a comma-separated values (csv) file
`pd.read_fwf(filepath_or_buffer[, colspecs, ...])`	Read a table of fixed-width formatted lines into DataFrame

Notes: the statements included in the 'Routine' column assume Pandas is loaded with the `pd` alias and there is a DataFrame loaded with name `df`.

Notes

[21]The set of competing frameworks include Python's PyTables and Dask, as well as R's data.tables and Julia's DataFrames.
[22]A time series is a set of points arranged in chronological order.
[23]Panel data, widely adopted in economics and finance, are multidimensional datasets wherein the same statistical units are repeatedly observed over time.
[24]Students interested in the origins of Pandas may want to read article on/interview with Wes McKinney "Meet the man behind the most important tool in data science."
[25]Pandas' API is well-documented and well-tested. Also, the user guide has plenty of examples showing Panda's features in action.
[26]In field of information systems and computer science, columns are often called fields; statisticians, economists, and analysts in general use the term variable to refer to a column; the term feature is common in the Machine Learning field.
[27]throughout the book, I refer to Python sessions without loss of generality. The points I make are valid also for IPython or Jupyter sessions.
[28]For the complete list of Pandas IO routines, please refer to `https://pandas.pydata.org/docs/reference/io.html`.

Chapter 6

Coda

In terms of *scope*, these notes emphasized the application of the Python Programming Language to 'data science' tasks. Mainly, we went through the basics of the language and considered two core modules: NumPy — focusing on technical and scientific computation — and Pandas — the *de facto* industry standard for data management.

In terms of *process*, learning the Python programming language takes months, if not years. However, it is possible to acquire a minimum understanding of Python in a few weeks. To do that, you should resist the temptation to search the web for the ultimate Python tutorial or off-the-shelf snippets that address your coding issues. Instead, you may want to spend time reading and digesting these notes offline. Why? Two arguments. First, learning a programming language requires scrutinizing and reasoning with coding snippets. Borrowing others' solutions helps to solve a problem but does not facilitate learning. Second, efficiency: the snippets reported in these notes have been selected to address the most common coding problems encountered in four previous Bayes Business Analytics MSc cohorts. So, you may not want to waste time on materials not relevant to the various Python-based modules offered at Bayes.

Finally, if you are a programming newbie, there will be times when you will find yourself asking questions such as: "What is the best way to do this?" or "How can I do this in Python?" Do not get frustrated — that is an integral part of the learning process!! Instead, keep calm and reach out to the lecturer and fellow students to share your thoughts and clarify your doubts.

Appendices

Appendix A

Cheat Sheets

A.1 Escapes

Table A.1
Helpful Escapes

Escape	Meaning
\\	Backslash (stores one \)
\'	Single quotes escape (stores ')
\"	Double quotes escape (stores ")
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

A.2 String Methods

TABLE A.2
Comprehensive List of String Methods

Cases I	
`s.capitalize()`	Capitalize s # 'hello' =>'Hello'
`s.lower()`	Lowercase s # 'HELLO' =>'hello'
`s.swapcase()`	Swap cases of all characters in s # 'Hello' =>"hELLO"
`s.title()`	Titlecase s # 'hello world' =>'Hello World'
`s.upper()`	Uppercase s # 'hello' =>'HELLO'
Sequence Operations I	
`s2 in s`	Return true if s contains s2
`s + s2`	Concat s and s2
`len(s)`	Length of s
`min(s)`	Smallest character of s
`max(s)`	Largest character of s
Sequence Operations II	
`s2 not in s`	Return true if s does not contain s2
`s * integer`	Return integer copies of s concatenated # 'hello' =>'hellohellohello'
`s[index]`	Character at index of s
`s[i:j:k]`	Slice of s from i to j with step k
`s.count(s2)`	Count of s2 in s
Whitespace I	
`s.center(width)`	Center s with blank padding of width # 'hi' =>' hi '
`s.isspace()`	Return true if s only contains whitespace characters
`s.ljust(width)`	Left justify s with total size of width # 'hello' =>'hello '
`s.rjust(width)`	Right justify s with total size of width # 'hello' =>' hello'
`s.strip()`	Remove leading and trailing whitespace from s # ' hello ' =>'hello'
Find / Replace I	
`s.index(s2, i, j)`	Index of first occurrence of s2 in s after index i and before index j
`s.find(s2)`	Find and return lowest index of s2 in s
`s.index(s2)`	Return lowest index of s2 in s (but raise ValueError if not found)
`s.replace(s2, s3)`	Replace s2 with s3 in s
`s.replace(s2, s3, count)`	Replace s2 with s3 in s at most count times
`s.rfind(s2)`	Return highest index of s2 in s
`s.rindex(s2)`	Return highest index of s2 in s (raise ValueError if not found)
Cases II	
`s.casefold()`	Casefold s (aggressive lowercasing for caseless matching) # 'ßorat' =>'ssorat'
`s.islower()`	Return true if s is lowercase
`s.istitle()`	Return true if s is titlecased # 'Hello World' =>true
`s.isupper()`	Return true if s is uppercase

TABLE A.2
(Cont'ed)

Inspection I	
`s.endswith(s2)`	Return true if s ends with s2
`s.isalnum()`	Return true if s is alphanumeric
`s.isalpha()`	Return true if s is alphabetic
`s.isdecimal()`	Return true if s is decimal
`s.isnumeric()`	Return true if s is numeric
`s.startswith(s2)`	Return true is s starts with s2
Splitting I	
`s.join('123')`	Return s joined by iterable '123' # 'hello' =>'1hello2hello3'
`s.partition(sep)`	Partition string at sep and return 3-tuple with part before, the sep itself, and part after # 'hello' =>('he', 'l', 'lo')
`s.rpartition(sep)`	Partition string at last occurrence of sep, return 3-tuple with part before, the sep, and part after # 'hello' =>('hel', 'l', 'o')
`s.rsplit(sep, maxsplit)`	Return list of s split by sep with rightmost maxsplits performed
`s.split(sep, maxsplit)`	Return list of s split by sep with leftmost maxsplits performed
`s.splitlines()`	Return a list of lines in s # 'hello\nworld' =>['hello', 'world']
Inspection II	
`s[i:j]`	Slice of s from i to j
`s.endswith((s1, s2, s3))`	Return true if s ends with any of string tuple s1, s2, and s3
`s.isdigit()`	Return true if s is digit
`s.isidentifier()`	Return true if s is a valid identifier
`s.isprintable()`	Return true is s is printable
Whitespace II	
`s.center(width, pad)`	Center s with padding pad of width # 'hi' =>'padpadhipadpad'
`s.expandtabs(integer)`	Replace all tabs with spaces of tabsize integer # 'hello\tworld' =>'hello world'
`s.lstrip()`	Remove leading whitespace from s # ' hello ' =>'hello '
`s.rstrip()`	Remove trailing whitespace from s # ' hello ' =>' hello'
`s.zfill(width)`	Left fill s with ASCII '0' digits with total length width # '42' =>'00042'

A.3 NumPy Array Manipulation Routines

TABLE A.3
NumPy Universal Functions: Mathematical Operations

Universal Function	Synopsis
`add(x1, x2, /[, out, where, casting, order, ...])`	Add arguments element-wise
`subtract(x1, x2, /[, out, where, cas:ting, ...])`	Subtract arguments, element-wise
`multiply(x1, x2, /[, out, where, castin:g, ...])`	Multiply arguments element-wise
`matmul(x1, x2, /[, out, casting, order, ...])`	Matrix product of two arrays
`divide(x1, x2, /[, out, where, casting, ...])`	Divide arguments element-wise
`logaddexp(x1, x2, /[, out, where, casting, ...])`	Logarithm of the sum of exponentiations of the inputs
`logaddexp2(x1, x2, /[, out, where, casting, ...])`	Logarithm of the sum of exponentiations of the inputs in base-2
`true_divide(x1, x2, /[, out, where, ...])`	Divide arguments element-wise
`floor_divide(x1, x2, /[, out, where, ...])`	Return the largest integer smaller or equal to the division of the inputs
`negative(x, /[, out, where, casting, order, ...])`	Numerical negative, element-wise
`positive(x, /[, out, where, casting, order, ...])`	Numerical positive, element-wise
`power(x1, x2, /[, out, where, casting, ...])`	First array elements raised to powers from second array, element-wise
`float_power(x1, x2, /[, out, where, ...])`	First array elements raised to powers from second array, element-wise
`remainder(x1, x2, /[, out, where, casting, ...])`	Returns the element-wise remainder of division
`mod(x1, x2, /[, out, where, casting, order, ...])`	Returns the element-wise remainder of division
`fmod(x1, x2, /[, out, where, casting, ...])`	Returns the element-wise remainder of division
`divmod(x1, x2[, out1, out2], / [[, out, ...])`	Return element-wise quotient and remainder simultaneously
`absolute(x, /[, out, where, casting, order, ...])`	Calculate the absolute value element-wise
`fabs(x, /[, out, where, casting, order, ...])`	Compute the absolute values element-wise
`rint(x, /[, out, where, casting, order, ...])`	Round elements of the array to the nearest integer
`sign(x, /[, out, where, casting, order, ...])`	Returns an element-wise indication of the sign of a number
`heaviside(x1, x2, /[, out, where, casting, ...])`	Compute the Heaviside step function
`conj(x, /[, out, where, casting, order, ...])`	Return the complex conjugate, element-wise
`conjugate(x, /[, out, where, casting, ...])`	Return the complex conjugate, element-wise
`exp(x, /[, out, where, casting, order, ...])`	Calculate the exponential of all elements in the input array
`exp2(x, /[, out, where, casting, order, ...])`	Calculate 2**p for all p in the input array
`log(x, /[, out, where, casting, order, ...])`	Natural logarithm, element-wise
`log2(x, /[, out, where, casting, order, ...])`	Base-2 logarithm of x
`log10(x, /[, out, where, casting, order, ...])`	Return the base 10 logarithm of the input array, element-wise
`expm1(x, /[, out, where, casting, order, ...])`	Calculate exp(x) - 1 for all elements in the array
`sqrt(x, /[, out, where, casting, order, ...])`	Return the non-negative square-root of an array, element-wise

TABLE A.4
NumPy Universal Functions: Trigonometric Operations

Universal Function	Synopsis
`sin(x, /[, out, where, casting, order, ...])`	Trigonometric sine, element-wise
`cos(x, /[, out, where, casting, order, ...])`	Cosine element-wise
`tan(x, /[, out, where, casting, order, ...])`	Compute tangent element-wise
`arcsin(x, /[, out, where, casting, order, ...])`	Inverse sine, element-wise
`arccos(x, /[, out, where, casting, order, ...])`	Trigonometric inverse cosine, element-wise
`arctan(x, /[, out, where, casting, order, ...])`	Trigonometric inverse tangent, element-wise
`arctan2(x1, x2, /[, out, where, casting, ...])`	Element-wise arc tangent of x1/x2 choosing the quadrant correctly
`hypot(x1, x2, /[, out, where, casting, ...])`	Given the "legs" of a right triangle, return its hypotenuse
`sinh(x, /[, out, where, casting, order, ...])`	Hyperbolic sine, element-wise
`cosh(x, /[, out, where, casting, order, ...])`	Hyperbolic cosine, element-wise
`tanh(x, /[, out, where, casting, order, ...])`	Compute hyperbolic tangent element-wise
`arcsinh(x, /[, out, where, casting, order, ...])`	Inverse hyperbolic sine element-wise
`arccosh(x, /[, out, where, casting, order, ...])`	Inverse hyperbolic cosine, element-wise
`arctanh(x, /[, out, where, casting, order, ...])`	Inverse hyperbolic tangent element-wise
`degrees(x, /[, out, where, casting, order, ...])`	Convert angles from radians to degrees
`radians(x, /[, out, where, casting, order, ...])`	Convert angles from degrees to radians
`deg2rad(x, /[, out, where, casting, order, ...])`	Convert angles from degrees to radians
`rad2deg(x, /[, out, where, casting, order, ...])`	Convert angles from radians to degrees

TABLE A.5
NumPy Universal Functions: Floating Operations

Universal Function	Synopsis
`isfinite(x, /[, out, where, casting, order, ...])`	Test element-wise for finiteness (not infinity and not Not a Number)
`isinf(x, /[, out, where, casting, order, ...])`	Test element-wise for positive or negative infinity
`isnan(x, /[, out, where, casting, order, ...])`	Test element-wise for NaN and return the result as a boolean array
`isnat(x, /[, out, where, casting, order, ...])`	Test element-wise for NaT (not a time) and return the result as a boolean array
`fabs(x, /[, out, where, casting, order, ...])`	Compute the absolute values element-wise
`signbit(x, /[, out, where, casting, order, ...])`	Returns element-wise True where signbit is set (less than zero)
`copysign(x1, x2, /[, out, where, casting, ...])`	Change the sign of x1 to that of x2, element-wise
`nextafter(x1, x2, /[, out, where, casting, ...])`	Return the next floating-point value after x1 towards x2, element-wise
`spacing(x, /[, out, where, casting, order, ...])`	Return the distance between x and the nearest adjacent number
`modf(x[, out1, out2], / [[, out, where, ...])`	Return the fractional and integral parts of an array, element-wise
`ldexp(x1, x2, /[, out, where, casting, ...])`	Returns x1 * 2**x2, element-wise
`frexp(x[, out1, out2], / [[, out, where, ...])`	Decompose the elements of x into mantissa and twos exponent
`fmod(x1, x2, /[, out, where, casting, ...])`	Returns the element-wise remainder of division
`floor(x, /[, out, where, casting, order, ...])`	Return the floor of the input, element-wise
`ceil(x, /[, out, where, casting, order, ...])`	Return the ceiling of the input, element-wise
`trunc(x, /[, out, where, casting, order, ...])`	Return the truncated value of the input, element-wise

TABLE A.6
NumPy Universal Functions: Comparison Operations

Universal Function	Synopsis
`greater(x1, x2, /[, out, where, casting, ...])`	Return the truth value of (x1 ¿ x2) element-wise
`greater_equal(x1, x2, /[, out, where, ...])`	Return the truth value of (x1 ¿= x2) element-wise
`less(x1, x2, /[, out, where, casting, ...])`	Return the truth value of (x1 ¡ x2) element-wise
`less_equal(x1, x2, /[, out, where, casting, ...])`	Return the truth value of (x1 ¡= x2) element-wise
`not_equal(x1, x2, /[, out, where, casting, ...])`	Return (x1 != x2) element-wise
`equal(x1, x2, /[, out, where, casting, ...])`	Return (x1 == x2) element-wise
`logical_and(x1, x2, /[, out, where, ...])`	Compute the truth value of x1 AND x2 element-wise
`logical_or(x1, x2, /[, out, where, casting, ...])`	Compute the truth value of x1 OR x2 element-wise
`logical_xor(x1, x2, /[, out, where, ...])`	Compute the truth value of x1 XOR x2, element-wise
`logical_not(x, /[, out, where, casting, ...])`	Compute the truth value of NOT x element-wise
`maximum(x1, x2, /[, out, where, casting, ...])`	Element-wise maximum of array elements
`minimum(x1, x2, /[, out, where, casting, ...])`	Element-wise minimum of array elements
`fmax(x1, x2, /[, out, where, casting, ...])`	Element-wise maximum of array elements
`fmin(x1, x2, /[, out, where, casting, ...])`	Element-wise minimum of array elements

TABLE A.7
NumPy Array Manipulation Routines

Routine	Synopsis
Basic operations	
`np.copyto(dst, src[, casting, where])`	Copies values from one array to another, broadcasting as necessary
`np.shape(a)`	Return the shape of an array
Changing array shape	
`np.reshape(a, newshape[, order])`	Gives a new shape to an array without changing its data
`np.ravel(a[, order])`	Return a contiguous flattened array
`np.ndarray.flat`	A 1-D iterator over the array
`np.ndarray.flatten([order])`	Return a copy of the array collapsed into one dimension
Changing array shape	
`np.moveaxis(a, source, destination)`	Move axes of an array to new position
`np.rollaxis(a, axis[, start])`	Roll the specified axis backwards, until it lies in a given position
`np.swapaxes(a, axis1, axis2)`	Interchange two axes of an array
`np.ndarray.T`	The transposed array
`np.transpose(a[, axes]`	Reverse or permute the axes of an array; returns the modified array
Changing number of dimensions	
`np.atleast_1d(*arys)`	Convert inputs to arrays with at least one dimension
`np.atleast_2d(*arys)`	View inputs as arrays with at least two dimensions
`np.atleast_3d(*arys)`	View inputs as arrays with at least three dimensions
`np.broadcast`	Produce an object that mimics broadcasting
`np.broadcast_to(array, shape[, subok])`	Broadcast an array to a new shape
`np.broadcast_arrays(*args[, subok])`	Broadcast any number of arrays against each other
`np.expand_dims(a, axis)`	Expand the shape of an array
`np.squeeze(a[, axis])`	Remove axes of length one from a

Table A.7 (cont'd)

Changing kind of array	
`np.asarray(a[, dtype, order, like])`	Convert the input to an array
`np.asanyarray(a[, dtype, order, like])`	Convert the input to an ndarray, but pass ndarray subclasses through
`np.asmatrix(data[, dtype])`	Interpret the input as a matrix
`np.asfarray(a[, dtype])`	Return an array converted to a float type
`np.asfortranarray(a[, dtype, like])`	Return an array (ndim >= 1) laid out in Fortran order in memory
`np.ascontiguousarray(a[, dtype, like])`	Return a contiguous array (ndim >= 1) in memory (C order)
`np.asarray_chkfinite(a[, dtype, order])`	Convert the input to an array, checking for NaNs or Infs
`np.require(a[, dtype, requirements, like])`	Return an ndarray of the provided type that satisfies requirements
Joining arrays	
`np.concatenate([axis, out, dtype, casting])`	Join a sequence of arrays along an existing axis
`np.stack(arrays[, axis, out])`	Join a sequence of arrays along a new axis
`np.block(arrays)`	Assemble an nd-array from nested lists of blocks
`np.vstack(tup)`	Stack arrays in sequence vertically (row wise)
`np.hstack(tup)`	Stack arrays in sequence horizontally (column wise)
`np.dstack(tup)`	Stack arrays in sequence depth wise (along the third axis)
`np.column_stack(tup)`	Stack 1-D arrays as columns into a 2-D array
`np.row_stack(tup)`	Stack arrays in sequence vertically (row wise)
Splitting arrays	
`np.split(ary, indices_or_sections[, axis])`	Split an array into multiple sub-arrays as views into array
`np.array_split(ary, indices_or_sections[, axis])`	Split an array into multiple sub-arrays
`np.dsplit(ary, indices_or_sections)`	Split array into multiple sub-arrays along the 3rd axis (depth)
`np.hsplit(ary, indices_or_sections)`	Split an array into multiple sub-arrays horizontally (column-wise)
`np.vsplit(ary, indices_or_sections)`	Split an array into multiple sub-arrays vertically (row-wise)

TABLE A.7 (cont'd)

Tiling elements	
`np.tile(A, reps)`	Construct an array by repeating A the number of times given by reps
`np.repeat(a, repeats[, axis])`	Repeat elements of an array
Adding and removing elements	
`np.delete(arr, obj[, axis])`	Return a new array with sub-arrays along an axis deleted
`np.insert(arr, obj, values[, axis])`	Insert values along the given axis before the given indices
`np.append(arr, values[, axis])`	Append values to the end of an array
`np.resize(a, new_shape)`	Return a new array with the specified shape
`np.trim_zeros(filt[, trim])`	Trim the leading and/or trailing zeros from a 1-D array or sequence
`np.unique(ar[, return_index, return_inverse, ...])`	Find the unique elements of an array
Rerraning elements	
`np.flip(m[, axis])`	Reverse the order of elements in an array along the given axis
`np.fliplr(m)`	Reverse the order of elements along axis 1 (left/right)
`np.flipud(m)`	Reverse the order of elements along axis 0 (up/down)
`np.reshape(a, newshape[, order])`	Gives a new shape to an array without changing its data
`np.roll(a, shift[, axis])`	Roll array elements along a given axis
`np.rot90(m[, k, axes])`	Rotate an array by 90 degrees in the plane specified by axes

Notes: the statements included in the 'Routine' column assume NumPy is loaded with the `np` alias.

Appendix B

GitHub: The World-Leading Collaborative and Versioning Tool

B.1 GitHub in a Nuthshell

GitHub is one of the most prominent Internet hosting service for software development and version control using Git, the free and OSS for distributed version control. Mainly, GitHub allows to:

- Host and manage repositories containing source code
- Share a repository with other users, publicly or privately
- Track changes to the source code
- Collaborate with other users to develop the source code

B.2 GitHub in the Education Sector

Since the 2018/19 academic year, my Python-related teaching has been based on GitHub. According to recent research, GitHub has many distinctive features (see Figure B.1) that make it a valuable teaching tool in teaching quantitative business subjects. Particularly, Zagalsky and colleagues[1] identify the following benefits of using GitHub for education:

- Transparency of activities
- Encouraging learners' participation
- Industry relevance
- Ease of use
- Free academic licensing
- Shared spaces and course versioning

Considering the characteristics of many Bayes Business Analytics modules, I think GitHub offers several advantages. *First*, it allows lecturers to share a substantial volume of computer code efficiently and transparently without creating a sense of information overload for learners. Below is a quote regarding the module 'SMM638, Network Analytics' offered in the 2019-20 academic year, when I started to use GitHub in my teaching:

> *"I think GitHub is a very efficient platform for our study. It's mainly because our teacher always updates all information in advance, so we can easily find everything we want."*

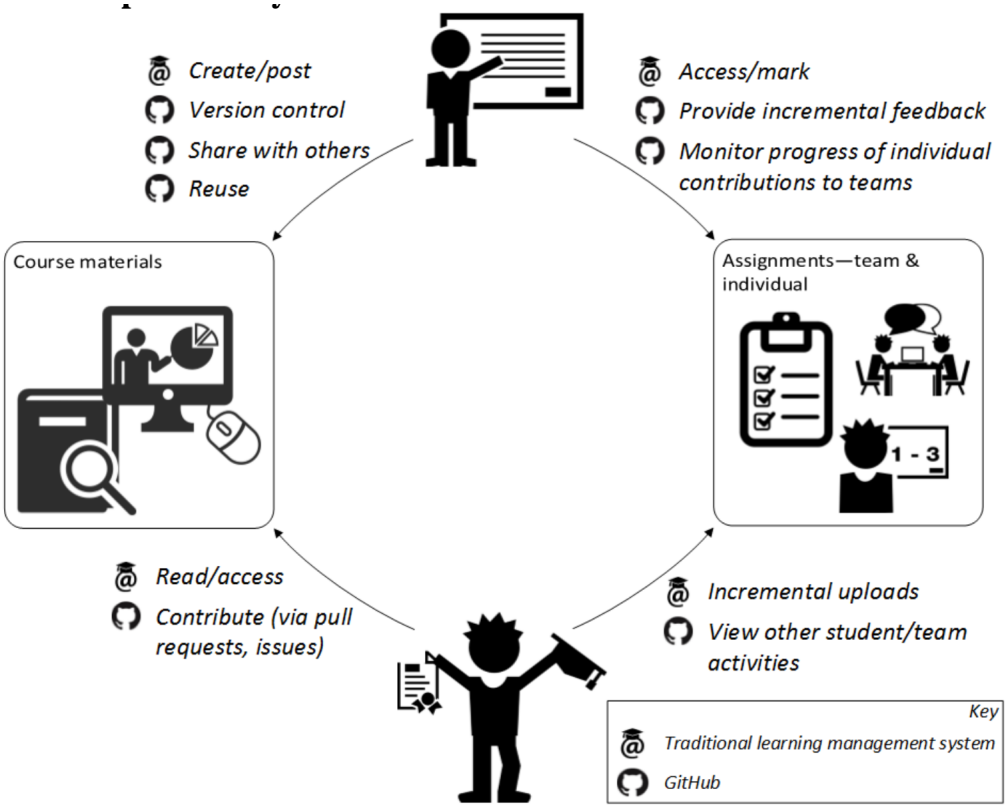


Figure B.1: The distinctive features of GitHub *vis a' vis* traditional LMS
Notes. — Source is Zagalsky et al. 2015.

Unlike a traditional Moodle page, GitHub has a very granular and public logging system and visualizations that highlight any change in the set of materials hosted in a repository (see, for example, Figure B.2). Furthermore, learners can decide how and to what extent to receive notifications on the changes affecting a GitHub repository.[2] Regarding 'SMM694, Applied Natural Language Processing' — delivered between May and July 2022 — 15 students out of 67 (22%) decided 'to watch' the GitHub repository hosting my teaching materials. Hence, they received real-time notifications on any edits of the 81 commits I made in the Summer Term;[3] 43 students out of 67 (64%) added my GitHub repository to their library; 31 out of 67 (42%) reused my repository's material for a new project.

Second, by engaging with GitHub, learners familiarise themselves with the industry-standard versioning software and collaborative platform. *Third*, GitHub is a piece of open-source and free software. As such, it presents no entry barriers for learners and can be deployed to sustain inclusive forms of teaching over and beyond the boundaries of individual academic institutions.[4] According to the tracking system of GitHub, as of today, there are 154 projects (a.k.a., forks) that build on my teaching materials covering Python, network analytics, data visualization, and NLP. Finally, in the case of hybrid teaching, the public logging system of GitHub provides the group of face-to-face learners and the group of remote learners with equivalent information about the functioning of the module and teaching materials. In other words, GitHub can prevent the formation of information asymmetry that potentially penalize remote learners because of their lack of physical proximity to the lecturer and fellow learners.

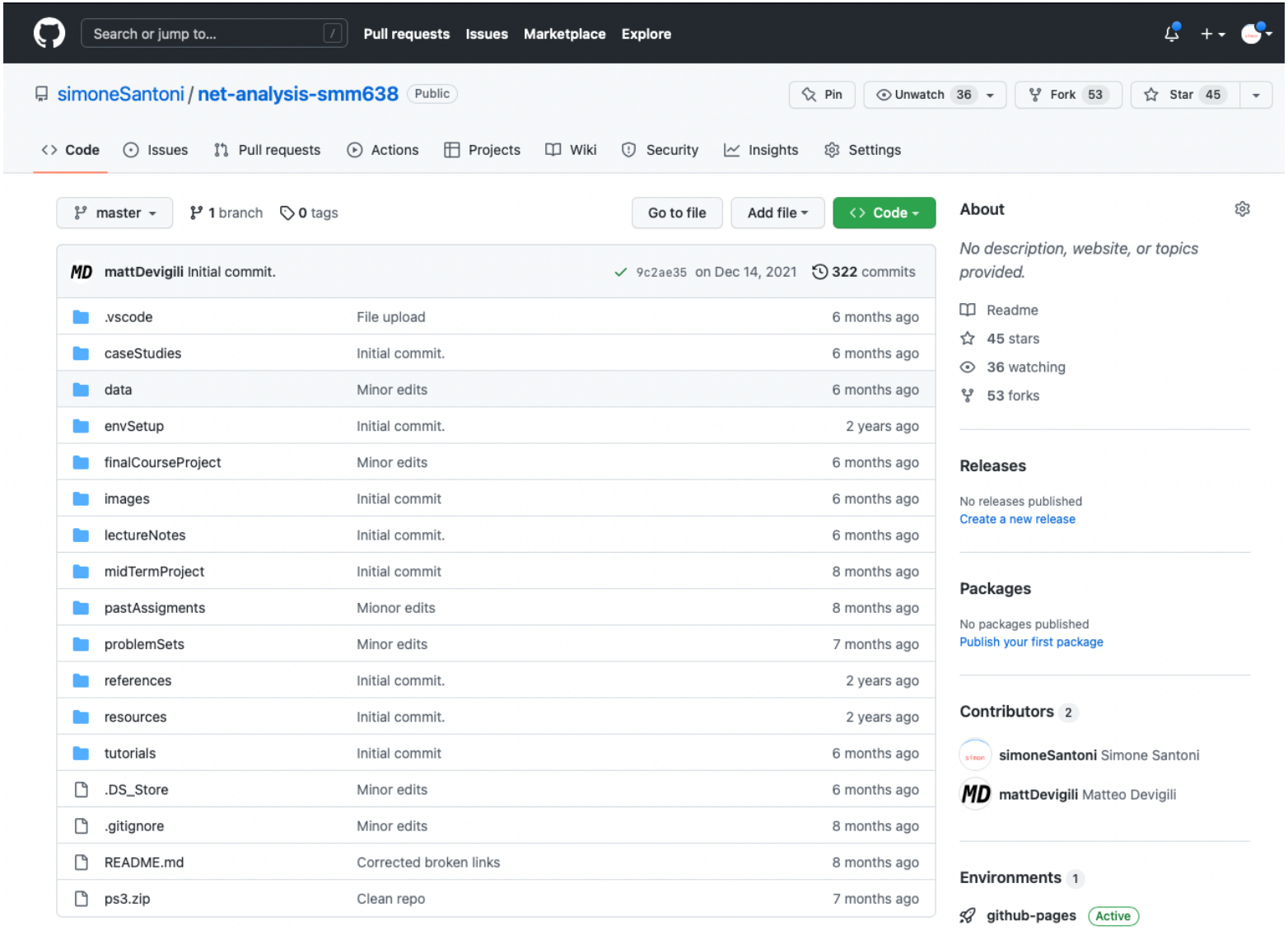


Figure B.2: A screenshot of the GitHub repository behind SMM638, Network analytics

Notes. — See https://github.com/simoneSantoni/net-analysis-smm638

.

B.3 Getting Started with GitHub

To work with GitHub, you need to have an account. You can create one for free at https://github.com. Programming newbies may want to interact with GitHub through GitHub Desktop (see Figure B.3) rather than through the command line. The graphical installer of GitHub Desktop is available at https://desktop.github.com/downloads and is free to use.

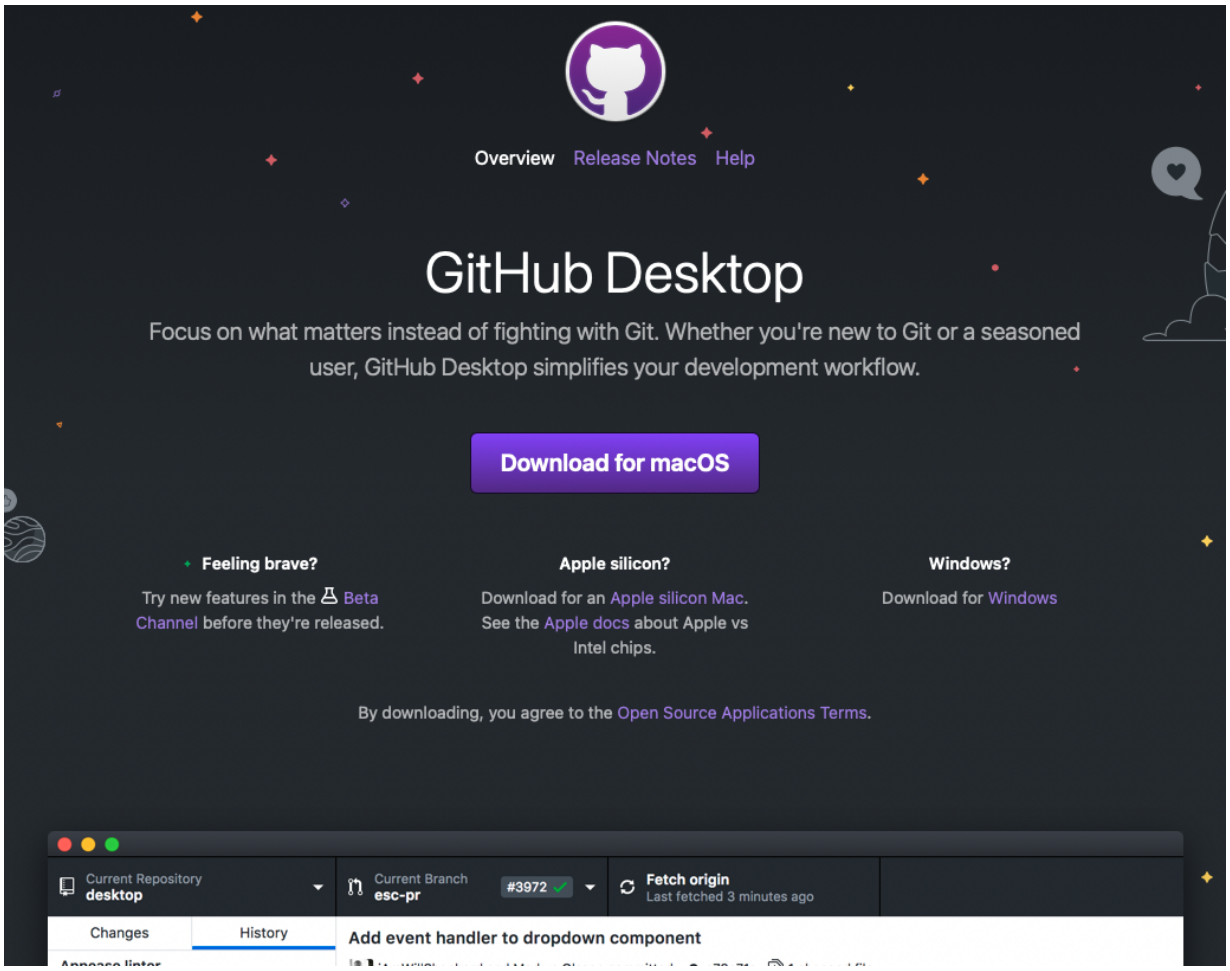


Figure B.3: A screenshot of the GitHub desktop application webpage

Once GitHub Desktop is installed on your computer, you have to log in to your account. You can do this via the 'Preferences' menu (see Figure B.4). At this point, it is possible to interact with GitHub. For example, you may 'clone' a repository hosted on GitHub. To do that, you search for a repository name (see Figure B.5) or pass the URL of the repository (see Figure B.6). Then, you have a local copy of the repository at the exact moment in time when you cloned it.

GihubDesktop has point-and-click structure (see Figure B.7):

- The left-hand side button, named 'Current Repository,' allows one to select the cloned repository one wants to work with
- The central button, named 'Current Branch,' allows one to select the branch one wants to work with. If you are not familiar with GitHub or similar software, you may want to know a branch is a version of a repository. For example, the 'master' branch is the default branch of a repository. It is the branch that contains the most recent version of the repository
- The right-hand side button, named 'Fetch Origin,' allows one to synchronize the local clone of the repository with the remote version of the repository
- The left-hand sidebar has two boxes:

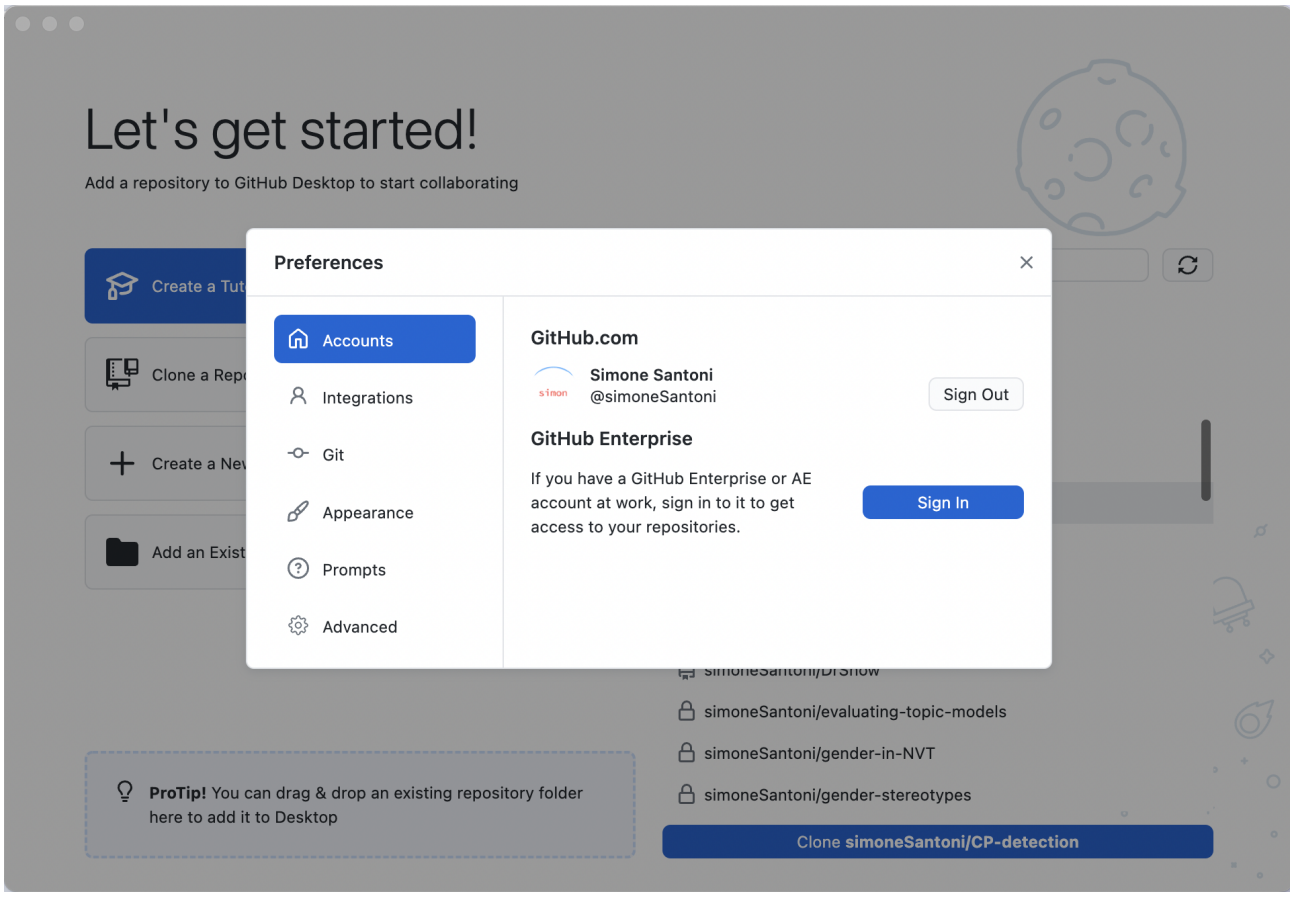


Figure B.4: A screenshot showing how to login into a GitHub account from within GitHub Desktop

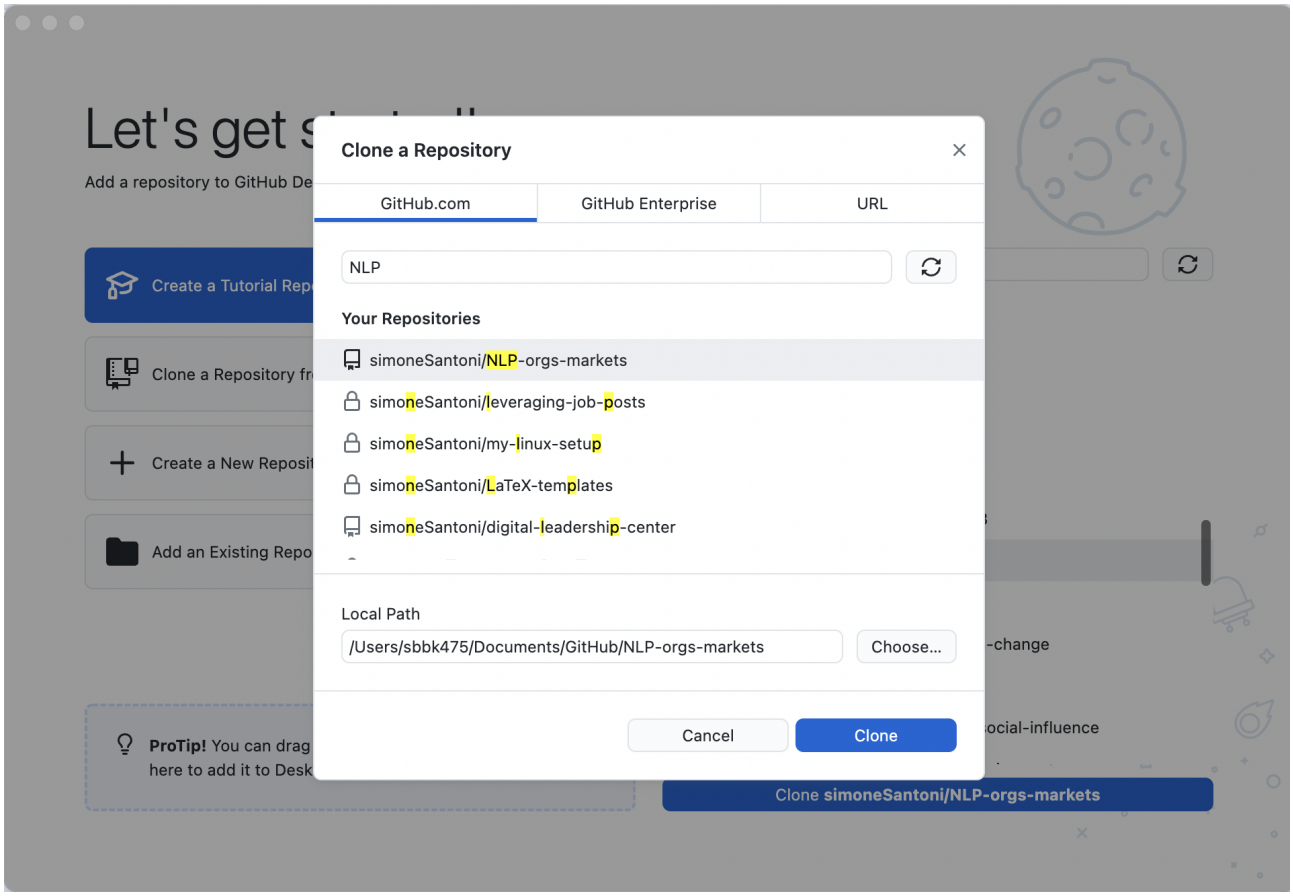


Figure B.5: A screenshot showing how to search for a repository name and then clone it

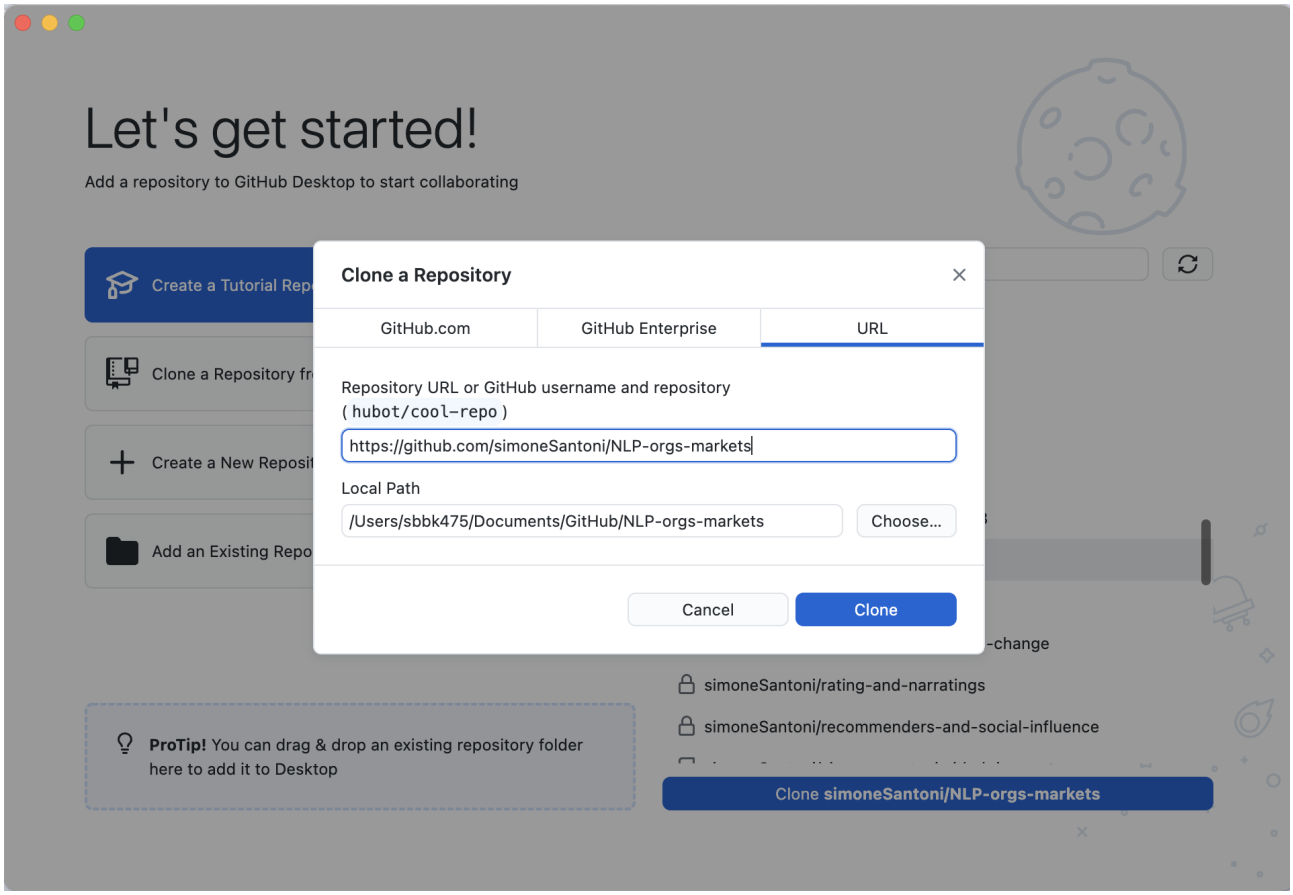


Figure B.6: A screenshot showing how to clone a repo using its URL

- The top box has two tabs:
  - 'History' reports the history of the changes made to the repository
  - 'Changes' reports the changes made to the local clone of the repository
- The bottom box contains a text input area in which one can write a commit message (e.g., 'added a new file') and the button to commit the changes made to the remote version of the repository (provided you are the owner of the repository or you have been granted the permission to do so)

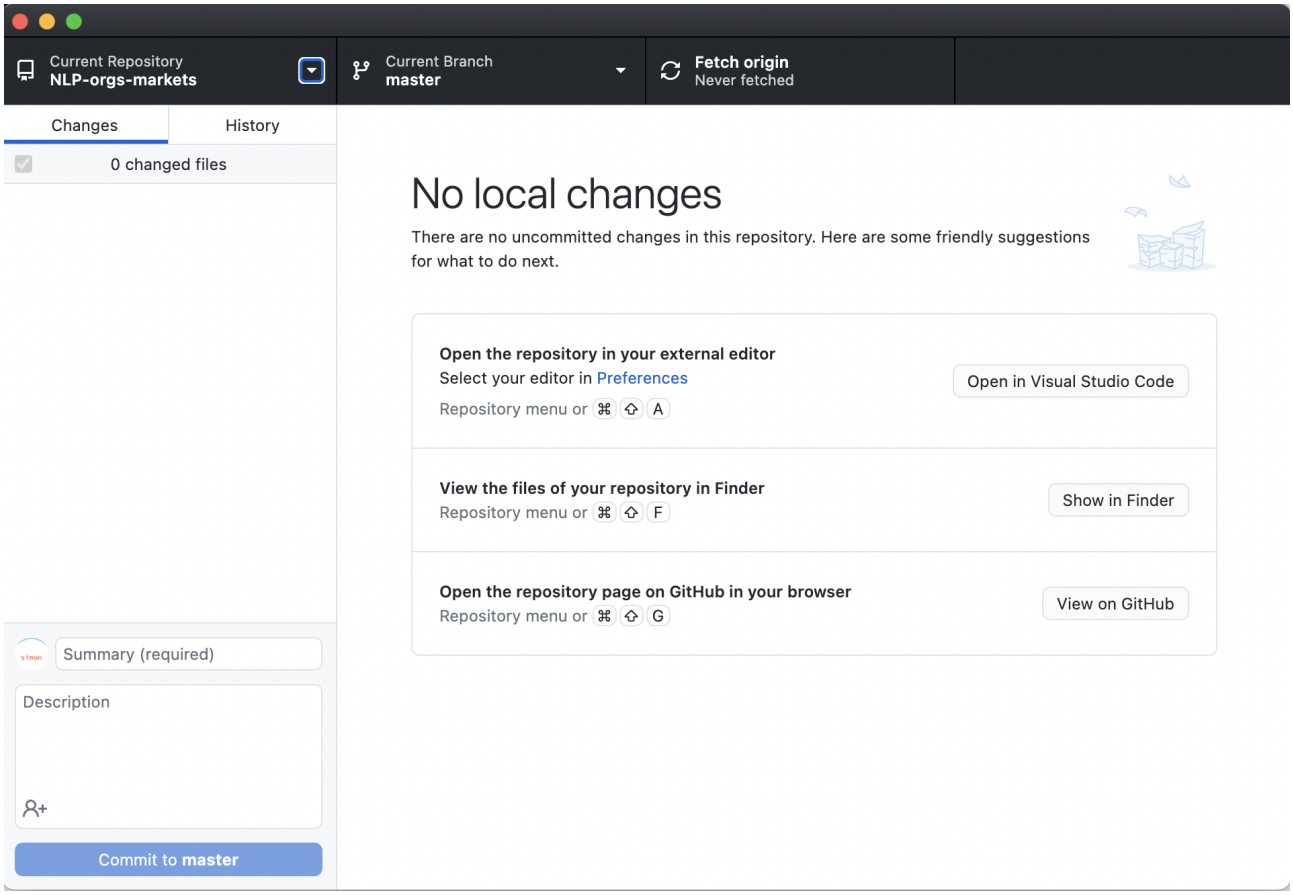


Figure B.7: A screenshot showing the actions available through GitHub Desktop

Notes

[1]Zagalsky, Alexey, Joseph Feliciano, Margaret-Anne Storey, Yiyun Zhao, and Weiliang Wang. "The emergence of GitHub as a collaborative platform for education" In *Proceedings of the 18th ACM conference on computer supported cooperative work social computing*, pp. 1906-1917. 2015.
[2]Especially, GitHub users can fine-tune the amount of information to receive by selecting the following options: i) by following the user owning a repository, a learner gets their GitHub homepage with the recent activities performed by that user; ii) by starring a repository, a learner has access to shortcuts to quickly and conveniently access a GitHub repository; iii) by watching a repository, a learner receives emails regarding any edit regarding a GitHub repository.
[3]The inter-temporal distribution of commits regarding SMM694 is available at
https://github.com/simoneSantoni/NLP-orgs-markets/graphs/commit-activity
[4]Iiyoshi, Toru, and M. S. V. Kumar. *Opening up education: The collective advancement of education through open technology, open content, and open knowledge*. The MIT Press, 2010.