

Learn Python Programming - 3rd Edition - Packt

1. A Gentle Introduction to Python

- **A proper introduction:**

- ◇ The two main features any object has are properties and methods.
 - **Properties:** are characteristics of an object.
 - **Methods:** are things an object can do.
- ◇ Objects expose **methods** and that can be run and properties that you can inspect.
- ◇ According to the data model documentation on the official Python documentation:
 - **"Object's are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects."**
 - <https://docs.python.org/3/reference/datamodel.html>
- ◇ For now all we need to know is that **every object in Python has an ID (or identity), a type, and a value.**
- ◇ Once created, the ID of an object is never changed. It's a unique identifier for it, and it's used behind the scenes by Python to retrieve the object when we want to use it. The type also never changes. The type states what operations are supported by the object and the possible values that can be assigned to it. The value of data can be changed or not: if it can, the object is said to be **mutable**, otherwise, it is said to be **immutable**.
- ◇ An important aspect of Python is its intrinsic multiparadigm nature. You can use it as a scripting language, but you can also exploit object-oriented, imperative, and functional programming styles, it is extremely versatile.

- **About Virtual Environments:**

- ◇ When working with Python it is very common to use **virtual environments**. Let's see what they are and why we need them by means of a simple example:
 - You install Python on your system and you start working on a website for . You create a project folder and start coding. Along the way, you also install some libraries; for example the Django framework, let's say the Django version you install for **Project X** is **2.2**.
 - Now, your website is so good that you get another client, **Y**. This person wants you to build another website, so you start **Project Y** and, along the way, you need to install Django again. The only issue is that now the Django version is 3.0 and you cannot install it on your system because this would replace the version you installed for **Project X**. You don't want to risk introducing incompatibility issues, so you have two options: you either stick with the version you have currently on your machine, or you upgrade it and make sure the first project is still fully working with the new version.

- ◇ Virtual environments are isolated Python environments, each of which is a folder that

contains all the necessary executables to use the packages that a Python project would need.

◇ So, you create a virtual environment for Project X, install all the dependencies, and then you create a virtual environment for Project Y, installing all its dependencies, without the slightest worry that because every library you install ends up within the boundaries of the appropriate virtual environment. In our example, Project X will hold Django 2.2, while Project Y will hold Django 3.0.

◇ **Note:**

- It is of vital importance that you never install libraries at the system level. Linux for example, relies on Python for many different tasks and operations, and if you fiddle with the system installation of Python, you risk compromising the integrity of the whole system.

- So, take this as a rule, such as brushing your teeth, before going to bed:

- **always create a virtual environment when starting a new project.**

◇ When it comes to creating a virtual environment on your system, there are a few different methods to carry this out. As of **Python 3.5**, the suggested way to create a virtual environment is to use the venv module. You can look it up on the official documentation page for more information:

- <https://docs.python.org/3/library/venv.html>

=====

◇ **Creating virtual environments:**

1- First go to the directory where you wish to create the project

2- Second create a virtual environment with whatever name you want:

- Execute the **venv** command:

→

```
python3 -m venv <virtual_environment_name>
```

3- Then we have to activate the virtual environment (do this on powershell):

- `<virtual_environment_name>\Scripts\activate`

4- Now run:

- `where python`

→ and it should give you the environments Python directory.

5- Now type ' **Python** ' (without quotes) in the console and it should enter Python mode.

6- You can use **exit()** to exit the Python environment.

7- Use **deactivate** to deactivate the environment.

=====

- **Installing third-party libraries:**

- ◇ In order to install third-party libraries, we need to use the Python Package Installer, known as **pip**.

- ◇ You use the following command to install third-party libraries taken from a **requirements** file:

- `pip install -r requirements.txt`

- **How is Python code organized:**

- ◇ Python gives you a structure, called a **package**, which allows you to group modules together. A package is nothing more than a folder that must contain a special file:

- **`__init__.py`**

- This does not need to hold any code, but its presence is required to tell Python that this is not just a typical folder - it is actually a large package.

- ◇ Here's an example of how the **example** directory would look like:

- first we issue the following command:

- `tree example`

→

```
example
├── core.py
├── run.py
└── util
    ├── __init__.py
    ├── db.py
    ├── math.py
    └── network.py
```

⇒ Within the root of this example we have two modules, **core.py** and **run.py**, and one package, **util**.

- Within **core.py** there may be the core logic of our application.
 - Within the **run.py** module we can probably find the logic to start the application.

- In the **util** package we can find various utility tools, we can guess that the modules there are named based on the type of tools they hold.

- **How do we use modules and packages:**

- ◇ We basically use a bunch of functions so we don't repeat code.

- ◇ In the previous example:

- the package **util** is our **utility library**. This is our custom belt that holds all those reusable tools (that is, functions), which we need in our application. Some of them will use things that are

out of the scope of Python's standard library, so we have to code them ourselves.

- **Python's Execution Model:**

- ◊ **Names and namespaces:**

- **Python names** are the closest abstraction to what other languages call **variables**. **Names** basically refer to objects and are introduced by **name-binding** operations.

- Let's see a quick example:

→

```
>>> n = 3 # integer number
>>> address = "221b Baker Street, NW1 6XE, London" # Sherlock Holmes' address
>>> employee = {
...     'age': 45,
...     'role': 'CTO',
...     'SSN': 'AB1234567',
... }
>>> # let's print them
>>> n
3
>>> address
'221b Baker Street, NW1 6XE, London'
>>> employee
{'age': 45, 'role': 'CTO', 'SSN': 'AB1234567'}
>>> other_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'other_name' is not defined
>>>
```

⇒ Remember that each Python object has an **identity**, a **type**, and a **value**. We defined three objects in the preceding code; let's now examine their type and values:

- An **integer** number **n** (**type: int, value: 3**)
 - A **string** **address** (**type: str, value: Sherlock Holmes' address**)
 - A **dictionary** **employee** (**type: dict, value: a dictionary object with three key/value pairs**).

- ◊ The previous objects are **names**:

- these can be used to retrieve data from within our code. They need to be kept somewhere so that whenever we need to retrieve those objects, we can use their names to fetch them. We need some space to hold them, hence the name: **namespaces**.

- **Namespaces**: is a mapping **from names to objects**. Examples are the set of builtin names (containing functions that are always accessible in any Python program), the global names in a module, and the local names in a function. Even the set of attributes of an object can be

considered a namespace.

- Namespaces allow you to define and organize your names with clarity, without overlapping or interference. For example, the namespace associated with the book we were looking for (in a previous example) in the library can be used to import the book itself, like this:

→

```
from library.second_floor.section_X.row_three import book
```

⇒ We start from the library namespace, and by means of the dot (.) operator, we walk into that namespace. Within this namespace, we look for **second_floor**, and again we walk into it with the . operator. We then walk into **section_x**, and finally, within the last namespace, **row_three**, we find the name we were looking for: **book**.

- There is another concept, closely related to that of a namespace, which we're gonna see briefly: **scope**:

- The order in which the namespaces are scanned when looking for a name is **local, enclosing, global, built-in (LEGB)**.

- **Basically everything in Python is an object, so they deserve a bit more attention:**

- ◇ Most of what you will ever do, in Python, has to do with manipulating objects.

- ◇ We have already seen that objects are Python's abstraction for data. In fact, everything in Python is an object: numbers, strings (data structures that hold text), containers, collections, even functions. You can think of them as if they were boxes with at least three features: an ID (which is unique), a type, and a value.

- But how do they come to life? How do we create them? How do we write our own custom objects? The answer lies in one simple word: **classes**.

- Objects are, in fact, instances of classes. The beauty of Python is that classes are objects themselves, but let's not go down this road. It leads to one of the most advanced concepts of this language: **metaclasses**.

- ◇ **Classes**: an abstract set of features and characteristics that together form something.

- Classes are used to create objects.

- When you have your own bike, it's an instance of the bike class. Your bike is an object with its own characteristics and methods. You have your own bike, same class, but different instance. Every bike ever created in the world is an instance of the bike class.

- Let's see an example. We will write a class that defines a bike and create two bikes, one red and one blue:

→

```
# Let's define the class Bike
class Bike:
    def __init__(self, color, frame_material):
        self.color = color
        self.frame_material = frame_material
    def brake(self):
        print("Braking!")

# Let's create a couple instances
red_bike = Bike('Red', 'Carbon fiber')
```

```

blue_bike = Bike('Blue', 'Steel')

#Let's inspect the objects we have, instances of the Bike class
print(red_bike.color) # prints: red
print(red_bike.frame_material) # prints: Carbon fiber
print(blue_bike.color) # prints: Blue
print(blue_bike.frame_material) # prints: Steel

#Let's brake
red_bike.brake() # prints: Braking!

```

⇒ The first **method**:

- **`__init__`**

◇ is an **initializer**. It uses some Python magic to set up the objects with the values we pass when we create.

◇ **Note:** every method that has leading and trailing underscores, in Python, is called a **magic method**. **Magic methods are used by Python for a multitude of different purposes**, hence it's never a good idea to name a custom method using two leading and trailing underscores.

- **Guidelines for writing good code (according to PEP 8 → <https://peps.python.org/pep-0008/>):**

- ◇ **Maximum Line Length:**

- Limit all line to a maximum of 79 characters.
 - for documentation strings or comments, the line length should be limited to 72 characters.

- ◇ **Should a Line Break Before or After a Binary Operator:**

```


# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)

# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)

```

- ◇ **Source File Encoding:**

- In the standard library, non-UTF-8 encodings should be used only for test purposes. Use non-ASCII characters sparingly, preferably only to denote place and human names. If using non-ASCII characters as data, avoid noisy Unicode characters like

ASCII characters as data, avoid noisy Unicode characters like , and byte order marks.

◇ Imports:

- Imports should usually be on separate lines:

```
# Correct:
import os
import sys

# Wrong:
import sys, os
```

- It's okay to say this though:

→

```
# Correct:
from subprocess import Popen, PIPE
```

- Imports should be grouped in the following order:

- 1> Standard library imports.
- 2> Related third party imports.
- 3> Local application/library specific imports.

- You should put a blank line between each group of imports.

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on **sys.path**):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Wildcard imports (**from <module> import ***) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools.

◇ Module Level Dunder Names:

- Module level "dunders" (i.e. names with two leading and two trailing underscores) such as:

- `__all__`
- `__author__`
- `__version__`
- etc.

should be placed after the module docstring but before any import statements except from **__future__** imports. Python mandates that future-imports must appear in the module before any other code except documentation strings:

```
"""This is the example module.

This module does stuff.
"""
```



```

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys

```

◇ **When to Use Trailing Commas:**

▪ When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments, or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter:

```

# Correct:
FILES = [
    'setup.cfg',
    'tox.ini',
]
initialize(FILES,
           error=True,
)

# Wrong:
FILES = ['setup.cfg', 'tox.ini',]
initialize(FILES, error=True,)

```

◇ **Comments:**

▪ You should use two spaces after a sentence - ending period in multi-sentence comments, except after the final sentence.

▪ Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure the code will never be read by people who don't speak your language.

◇ **Naming Conventions:**

▪ The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent - nevertheless, here are the currently recommended naming standards:

- **Descriptive: Naming Styles:**

→ **Note:** When using acronyms in **CapitalizedWords**, capitalize all the letters of the acronym. thus **HTTPServerError** is better than **HttpServerError**.

▪ **Prescriptive: Naming Conventions:**

- **Names to avoid:** Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numeral one and zero, when tempted to use 'l', use 'L' instead.

▪ **Package and Module Names:**

- Modules should have short, all-lowercase names. underscores can be used in the module

name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

- When an extension modules written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. **_socket**)

- **Class Names:**

- Class names should normally use the **CapWords** convention.

- **Type Variable Names:**

- Names of type variable introduced in PEP 484 should normally use CapWords preferring short names: **T**, **AnyString**, **Num**.

- **Exception Names:**

- Because exception should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

- **Global Variable Names:**

- The conventions are about the same as those for functions.
 - Modules that are designed for use via **from M import *** should use the **__all__** mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do indicate these globals are "module non-public").

- **Function and Variable Names:**

- Function names should be lowercase, with words separated by underscores as necessary to improve readability.
 - Variable names follow the same convention as function names.

- **Function and Method Arguments:**

- Always use **self** for the first argument to instace methods.
 - Always use **cls** for the first argument to class methods.
 - If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus **__class** is better than **clss**. (Perhaps better is to avoid such classes by using a synonym).

- **Method Names and Instance Variables:**

- Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.
 - Use one leading underscore for non-public methods and instance variables.
 - To avoid name clashes with subclasses, use two leading underscores to invoke Python's names mangling rules.
 - Python mangles these names with the class name: if class Foo has an attribure names **__a**, it cannot be accessed by **Foo.__a** (an insistent user could still gain access by calling **Foo._Foo_a**). Generally double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.
 - **Note:** there is some controversy about the use of **__names**

▪ Constants:

- Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include:
 - MAX_OVERFLOW
 - TOTAL

▪ Designing for Inheritance:

- Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; It's easier to make it public later than to make a public attribute non-public.
- We don't use the term "private" here, since no attribute is really private in Python (without a generally necessary amount of work).

◇ Programming Recommendations:

- Use **is not** operator rather than **not ... is**. While both expressions are functionally identical, the former is more readable and preferred:

```
# Correct:
if foo is not None:
# Wrong:
if not foo is None:
```

- When implementing ordering operations with rich comparisons, it is best to implement all six operations (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`) rather than relying on other code to only exercise a particular comparison.

- Always use a `def` statement instead of an assignment statement that binds a lambda expression directly to an identifier:

```
# Correct:
def f(x): return 2*x

# Wrong:
f = lambda x: 2*x
```

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare **except:** clause:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

- a bare **except:** clause will catch **SystemExit** and **KeyboardInterrupt** exceptions, making it harder to interrupt a program with Control+C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use **except Exception:** (bare **except** is equivalent to **except BaseException:**).

- Additionally, for all try/except clauses, limit the **try** clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs:

```

# Correct:
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)

```

- When a resource is local to a particular section of code, use a **with** statement to ensure it is cleaned up promptly and reliably after use. A **try/finally** statement is also acceptable.
- Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources:

```

# Correct:
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
# Wrong:
with conn:
    do_stuff_in_transaction(conn)

```

→ the latter example doesn't provide information to indicate that the **__enter__** and **__exit__** methods are doing something other than closing the connection after a transaction. Being explicit is important in this case

- Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as their **return None**, and an explicit return statement should be present at the end of the function (if reachable.):

```

# Correct:

def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
# Wrong:

def foo(x):
    if x >= 0:

```

```

        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)

```

- Use '**.startswith()**' and '**.endswith()**' instead of string slicing to check for prefixes or suffixes:

- '**.startswith()**' and '**.endswith()**' are cleaner and less error prone:

→

```

# Correct:
if foo.startswith('bar'):
# Wrong:
if foo[:3] == 'bar':

```

- Object type comparison should always use **isinstance()** instead of comparing types directly:

```

# Correct:
if isinstance(obj, int):
# Wrong:
- if type(obj) is type(1):

```

- For sequence, (strings, lists, tuples), use the fact that empty sequences are false:

```

# Correct:
if not seq:
if seq:

# Wrong:
if len(seq):
- if not len(seq):

```

- Don't compare boolean values to True or False using ==:

```

# Correct:
if greeting:

# Wrong:
- if greeting == True:

```

→ Worse:

```

# Wrong:
- if greeting is True:

```

- Use the flow control statements **return/break/continue** within the finally suite of a **try...finally**, where the flow control statements would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite:

```

# Wrong:
- def foo():

```

```
try:  
    1 / 0  
finally:  
    return 42
```

2. Built-In Data Types

- In this chapter we are going to cover the following:
 - ◇ Python's **objects' structures**.
 - ◇ **Mutability** and **inmutability**.
 - ◇ **Built-in data types: numbers, strings, dates and times, sequences, collection, and mapping types**.
 - ◇ The **collection** module.
 - ◇ **Enumerations**.

2.1 Everything is an object

- What really happens when you type an instruction like **age = 42** in a Python module.
- So, what happens is that an **object** is created. It gets an **id**, the **type** is set to **int**, and the **value** to **42**. A **name**, **age**, is placed in the **global namespace**, pointing to that object. Therefore, whenever we are in the **global namespace**, after the execution of that line, we can retrieve that object by simply accessing it through its name **age**.
- If you were to move house, you would put all the knives, forks, and spoons in a box and label it cutlery. This is exactly the same concept. Here is a screenshot of what it may look like:

◇



- So, for the rest of this chapter, whenever you read something such as **name = some_value**, think of a name placed in the namespace that is tied to the scope in which the instruction was written, with a nice arrow pointing to an object that has an **id**, a **type**, and a **value**.

2.2 Mutable or immutable? That is the question

- The first fundamental distinction that Python makes on data is about whether or not the value of an object can change. **If the value can change**, the object is called **mutable**, whereas **if the value cannot change**, the object is called **immutable**.
- It is very important that you understand the distinction between **mutable** and **immutable** because it affects the code you write; take this example:

```
>>> age = 42
>>> age
42
>>> age = 43 #A
>>> age
43
```

◊ In the preceding code, on line **#A**, have we changed the value of **age**? Well, no. But now it's **43**. Yes, it's 43, **but 42 was an integer number, of the type int, which is immutable**. So, what happened is really that on the first line, **age** is a name that is set to point to an **int** object, whose value is **42**. When we type **age = 43**, what happens is that another object is created, of the type **int** and value **43** (also, the **id** will be different), and the name **age** is set to point to it. So, in fact, we did not change that **42** to **43** - we actually just pointed **age** to a different location, which is the new **int** object whose value is **43**. let's see the same code also printing the IDs:

```
>>> age = 42
>>> id(age)
4377553168
>>> age = 43
>>> id(age)
4377553200
```

- Now, let's see the same example using a mutable object. For this example, let's just use a **Person** object, that has a property **age** (don't worry about the class declaration for now - it is there only for completeness):

◊

```

>>> class Person:
...     def __init__(self, age):
...         self.age = age
...
>>> fab = Person(age=42)
>>> fab.age
42
>>> id(fab)
4380878496
>>> id(fab.age)
4377553168
>>> fab.age = 25 # I wish!
>>> id(fab) # will be the same
4380878496
>>> id(fab.age) # will be different
4377552624

```

```

class Person:
    def __init__(self, age):
        self.age = age

fab = Person(age=42)

print('fab.age', '\n', fab.age)
print('id(fab)', '\n', id(fab))
print('id(fab.age)', '\n', id(fab.age))

fab.age = 25
print('id(fab)', '\n', id(fab))
print('id(fab.age)', '\n', id(fab.age))

```

▪ In this case we set up an object **fab** whose **type** is **Person** (a custom class). On creation, the object is given the age of 42. We then print it, along with the object **ID**, and the **ID of age** as well. Notice that, even after we change **age** to be **25**, the **ID** of **fab** stays the same (while the **ID** of **age** has changed, of course). **Custom objects in Python are mutable** (unless you code them not to be). Keep this concept in mind, as it's very important. We'll remind you about it throughout the rest of this chapter.

2.3 Numbers

- **Integers:**

- ◇ **Python integers have an unlimited range, subject only to available virtual memory.** This means that it doesn't really matter how big a number you want to store is, as long as it can fit in your computer's memory, Python will take care of it.

- ◇ Integer numbers can be positive, negative, or zero. They support all the basic mathematical operations, as shown in the following example:

```
a = 14
b = 3

a + b # addition
# result: 17

a - b # subtraction
# result: 11

a * b # multiplication
# result: 42

a / b # true division
# result: 4.666666666666667

a // b # integer division
# result: 4

a % b # modulo operation (remainder of division)
# result: 2

a ** b # power operation
# result: 2744
```

- ◇ Let's see how division behaves differently when we introduce negative numbers:

```
7 / 4 # true division
# result: 1.75

7 // 4 # integer division, truncation returns 1
# result: 1

-7 / 4 # true division again, result is opposite of previous
# result: -1.75

-7 // 4 # integer division, result not the opposite of previous
# result: -2
```

- This is an interesting example. If you were expecting a -1 on the last line, don't feel bad, it's just the way Python works. **Integer division in Python is always rounded towards minus infinity.** If, instead of flooring, you want to truncate a number to an integer, you can use the built-in **int()** function, as shown in the following example:

→

```
int(1.75)
# result: 1

int(-1.75)
# result: -1
```

⇒ **Notice that the truncation is done toward 0.**

- Note: the **int()** function can also return integer numbers from string representation in a given base:

◇

```
int('10110', base=2)
# result: 22
```

◇ The **pow()** function allows a third argument to perform modular **exponentiation**. The form with three arguments now accepts a negative exponent in the case where the base is relatively prime to the modulus.

- The result is the **modular multiplicative inverse** of the base (or a suitable power of that, when the exponent is negative, but not -1), modulo the third argument. Here's an example:

```
pow(123, 4)
# result: 228886641

pow(123, 4, 100)
# result: 41
# basically: 228886641 % 100 == 41

pow(37, -1, 43) # modular inverse of 37 mod 43
# result: 7

7 ** 37 % 43 # proof the above is correct
# result: 7
```

◇ One nice feature introduced in python 3.6 is the ability to add underscores within number literals (between digits or base specifiers, but not leading or trailing). The purpose is to help make some numbers more readable, such as **1_000_000_00**:

```
n = 1_024
n
# result:1024

hex_n = 0x4_0_0 # 0x400 == 1024
hex_n
# result: 1024
```

• Booleans:

◇ Booleans are a subclass of integers, so **True** and **False** behave respectively like **1** and **0**.

- **True** → **1**
- **False** → **0**

◇ Let's look at some examples:

▪

```
bool(-42)
# result: True # and so does every non-zero number

not True
# result: False

not False
# result: True

True and True
# result: True

False or True
# result: True
```

- You can see that **True** and **False** are **subclasses of integers** when you try to add them. Python upcasts them to integers and performs the addition:

→

```
1 + True
# result: 2

False + 42
# result: 42

7 - True
# result: 6
```

• Real numbers:

◇ Several programming languages give coders two different formats:

- **single** → **takes up 32 bits of memory**

→ and

- **double precision** → **takes up 64 bits of memory** (the only one supported by Python)

▪ The former takes up **32 bits** of memory, the latter **64**. Python supports only the double format.

◇ The **sys.float_info** sequence holds information about how floating point numbers will behave on your system. This is an example of what you might see:

▪

```
import sys
sys.float_info
# result: sys.float_info(max=1.7976931348623157e+308,
#                          max_exp=1024, max_10_exp=308,
#                          min=2.2250738585072014e-308,
#                          min_exp=-1021, min_10_exp=-307,
#                          dig=15, mant_dig=53,
#                          epsilon=2.220446049250313e-16,
#                          radix=2, rounds=1)
```

- Let's make a few considerations here:

→ We have 64 bits to represent floating point number. This means we can represent at most 2^{64} (that is **12,446,744,073,709,551,616**) distinct numbers. Take a look at the **max** and

epsilon values for the float numbers, and you will realize that it's impossible to represent them all. There is just not enough space, so they are approximated to the closest representable number. You probably think that only extremely big or extremely small numbers suffer from this issue. Well, think again and try the following in your console:

→

```
0.3 - 0.1 * 3 # this should be 0!!!  
# result: -5.551115123125783e-17
```

⇒ in the authors' system this yielded:

•

```
0.09
```

◇ What does this tell you? It tells you that double precision numbers suffer from approximation issues even when it comes to simple numbers like **0.1** or **0.3**. **Why is this important? It can be a big problem if you are handling prices, or financial calculations, or any kind of data that need not to be approximated.** Don't worry, Python gives you the **Decimal** type, which doesn't suffer from these issues. we'll see them in a moment.

• Complex numbers:

◇ Python gives you **complex numbers** support out of the box. Complex numbers are numbers that can be expressed in the form **a + ib**, where **a** and **b** are real numbers, and **i** (or **j** if you're an engineer) is the imaginary unit; that is the square root of **-1**. **a** and **b** are called, respectively, the **real** and **imaginary** part of the number.

◇ It is perhaps unlikely that you will use them, unless you're coding something scientific. Nevertheless, let's see a small example:

```
c = 3.14 + 2.73j  
c = complex(3.14, 2.73) # same as above  
c.real # real part  
# result: 3.14  
  
c.imag # imaginary part  
# result: 2.73  
  
c.conjugate() # conjugate of A + Bj is A - Bj  
# result: (3.14 - 2.73j)  
  
c * 2 # multiplication is allowed  
# result: (6.28 + 5.46j)  
  
c ** 2 # power operation as well  
# result: (2.4067000000000007+17.1444j)  
  
d = 1 + 1j # addition and subtraction as well  
c - d  
(2.14 + 1.73j)
```

• Fractions and decimals:

◇ Let's finish the tour of the number department with a look at fractions and decimals. Fractions hold a rational numerator and denominator in their lowest forms. Let's see an example:

```
from fraction import Fraction
Fraction(10, 6)
# result: Fraction(5, 3) # notice it's been simplified

Fraction(1, 3) + Fraction(2, 3) # 1/3 + 2/3 == 3/3 == 1/1
# result: Fraction(1, 1)

f = Fraction(10, 6)
f.numerator
# result: 5
f.denominator
# result: 3
f.as_integer_ratio()
# result: (5, 3)
```

◇ The **as_integer_ratio()** method has also been added to integers and Booleans. This is helpful, as it allows you to use it without needing to worry about what type of number is being worked with.

◇ Although **Fraction** objects can be very useful at times, it's not that common to spot them in commercial software. Instead, it is much more common to see decimal numbers being used in all those contexts where precision is everything:

- For example in scientific and financial calculations.

- **Note:** It's important to remember that arbitrary precision decimal numbers come at a price in terms of performance, of course. The amount of data to be stored for each number is greater than it is for **Fractions** or **floats**. The way they are handled also requires the Python interpreter to work harder behind the scenes. Another interesting thing to note is that you can get and set the precision by accessing **decimal.getcontext().prec** .

◇ Let's see a quick example with decimal numbers:

```
from decimal import Decimal as D # rename for brevity
D(3.14) # pi, from float, so approximation issues
# result: Decimal('3.140000000000000124344978758017532527446746826171875')

D('3.14') # pi, from a string, so no approximation issues
Decimal('3.14')

D(0.1) * D(3) - D(0.3) # from float, we still have the issue
# result: Decimal('2.775557561565156540423631668E-17')

D('0.1') * D(3) - D('0.3') # from string, all perfect
# result: Decimal('0.0')

D('1.4').as_integer_ratio() # 7/5 = 1.4
# result: (7, 5)
```

- **Notice that when we construct a Decimal number from a float, it takes on all the approximation issues a float may come with. On the other hand, when we create**

a Decimal from an integer or a string representation of a number, then the Decimal will have no approximation issues, and therefore no quirky behavior.

- **When it comes to currency or situations in which precision is of utmost importance, use decimals.**

2.4 Immutable Sequences

- Let's start with immutable sequences:

- ◇ **strings**
- ◇ **tuples**
- ◇ **bytes**

- **Strings and bytes:**

◇ Textual data in Python is handled with **str** objects, more commonly known as **strings**. **They are immutable sequences of Unicode code points**. Unicode code points can represent a character, but can also have other meanings such as when formatting, for example. Python, unlike other languages, doesn't have a **char** type, so a single character is rendered simply by a string of length 1.

◇ Unicode is an excellent way to handle data, and should be used for the internals of any application. When it comes to storing textual data though, or sending it on the network, you will likely want to encode it, using an appropriate encoding for the medium you are using. the result of an encoding produces a byte object, whose syntax and behavior is similar to that of strings. **String literals are written** in Python using **single, double, or triple quotes** (both single or double). If built with triple quotes, a string can span multiple lines, let's take a look at an example:

```
# 4 ways to make a string
str1 = 'String with single quotes.'
str2 = "String with double quotes."
str3 = '''String with multiple lines,
so it can span multiple lines.'''
str4 = """This too
is a multiline one
built with triple double-quotes."""

str4 # A
# result: 'This too\nis a multiline one\nbuilt with triple double-quotes.'

print(str4) # B
# result: This too is a multiline one
# built with triple double-quotes.
```

- In **#A** and **#B**, we print **str4**, first implicitly, and then explicitly, using the **print()** function. A good exercise would be to find out why they are different (look up the **str()** and **repr()** functions.)

- ◇ Strings, like any sequence, have a length. You can get this by calling the **len()** function:

```
len(str1)
# result: 26
```

◇ **Python 3.9** has introduced two new methods that deal with the prefixes and suffixes of strings. Here's an example that explains the way they work:

```
s = 'Hello There'
s.removeprefix('Hell')
# result: 'o There'

s.removesuffix('here')
# result: 'Hello T'

s.removeprefix('Oops')
# result: 'Hello There'
```

• **The nice thing about them is shown by the last instruction:** when we attempt to remove a prefix or suffix which is not there, the method simply returns a copy of the original string. This means that these methods, behind the scenes, are checking if the prefix or suffix matches the argument of the call, and when that's the case, they remove it.

• Encoding and decoding strings:

◊ Using the **encode** / **decode** methods, **we can encode Unicode strings and decode bytes objects**. **UTF-8** is a variable length **character encoding**, capable of encoding all possible Unicode code points. It is the most widely used encoding for the web. Notice also that by adding the literal **b** in front of a string declaration, we're creating a bytes object:

```
s = "This is üníc0de" # unicode string: code points
type(s)
# result: <class 'str'>

encoded_s = s.encode('utf-8') # utf-8 encoded version of s
encoded_s
# result: b'This is \xc3\xbc\xc5\x8b\xc3\xad0de' # result: bytes object

type(encoded_s) # another to verify it
# result: <class 'bytes'>

encoded_s.decode('utf-8') # let's revert to the original
# result: 'This is üníc0de'

bytes_obj = b"A bytes object" # a bytes object
type(bytes_obj)
# result: <class 'bytes'>
```

• Indexing and slicing strings:

◊ When manipulating sequences, it's very common to access them at one precise position (**indexing**), or to get a sub-sequence out of them (**slicing**). **When dealing with immutable sequences, both operations are read-only.**

◊ While indexing comes in one form (zero based access to any position within the sequence) slicing comes in different forms. When you get a slice of a sequence, you can specify the **start** and **stop** positions, along with the **step**. They are separated with a colon (:) like this:

- **my_sequence[start:stop:step]**

- All the arguments are optional; **start** is inclusive, and **stop** is exclusive. Let's take a look at an example:

```
s = "The trouble is you think you have time."
s[0] # Indexing at position 0, which is the first char
# result: 'T'

s[5] # indexing at position 5, which is the sixth char
# result: 'r'

s[:4] # slicing, we specify only the stop position
# result: 'The '

s[4:] # slicing, we specify only the start position
# result: 'trouble is you think you have time.'

s[2:14] # Slicing, both start and stop positions
# result: 'e trouble is'

s[2:14:3] # Slicing, start, stop and step (every 3 chars)
# result: 'erb '

s[:] # quick way of making a copy
# result: 'The trouble is you think you have time.'
```

→ The last line is quite interesting. If you don't specify any of the parameters, Python will fill in the defaults for you. In this case, **start** will be the start of the string, **stop** will be the end of the string, and **step** will be the default: **1**. This is an easy and quick way of obtaining a copy of the string **s** (the same value but a different object).

→ To get the reversed copy of a string using slicing:

⇒

```
s[::-1]
```

• String formatting:

◇ One of the features strings have **is the ability to be used as a template**. There are several different ways of formatting a string, and for the full list of possibilities, we encourage you to look up the documentation. Here are some common examples:

```
greet_old = 'Hello %s'
greet_old % 'Fabrizio!'
# result: 'Hello Fabrizio!'

greet_positional = 'Hello {}!'
greet_positional.format('Fabrizio!')
# result: 'Hello Fabrizio!'

greet_positional = 'Hello {} {}!'
greet_positional.format('Fabrizio', 'Romano')
# result: 'Hello Fabrizio Romano!'

greet_positional_idx = 'This is {0}! {1} loves {0}'
greet_positional_idx.format('Python!', 'Heinrich')
# result: 'This is Python! Heinrich loves Python!'

greet_positional_idx.format('Coffee!', 'Fab')
```

```
# result: 'This is Coffee! Fab loves Coffee!'

keyword = 'Hello, my name is {name} {last_name}'
keyword.format(name='Fabrizio', last_name='Romano')
# result: 'Hello, my name is Fabrizio Romano'
```

- In the previous example, you can see four different ways of formatting strings.
 - The first one, which relies on the **%** operator, is deprecated and shouldn't be used anymore.
 - The current, modern way to format a string is by using the **format()** string method. You can see, from the different examples, that a pair, of curly braces, acts as a placeholder within the string.
 - ⇒ When we call **format()**, we feed it data that replaces the placeholders. We can specify indexes (and much more) within the curly braces, and even names, which implies we'll have to **format()** using keyword arguments instead of positional ones.
- Notice how **greet_positional_idx** is rendered differently by feeding different data to the call to **format**.

◇ One last feature we'll take a look at was added to Python in version 3.6, and it's called **formatted string literals**. This feature is quite cool (and it is faster than using the **format()** method): strings are prefixed with **f**, and contain replacement fields surrounded by curly braces.

- Replacement fields are expressions evaluated at runtime, and then formatted using the format protocol:

```
name = 'Fab'
age = 42
f"Hello! My name is {name} and I'm {age}"
# result: "Hello! My name is Fab and I'm 42"

from math import pi
f"No arguing with {pi}, it's irrational..."
# result: "No arguing with 3.141592653589793, it's irrational..."
```

- An interesting addition to f-strings, which was introduced in Python 3.8, is the ability to add an equals sign specifier within the f-string clause; this causes the expression to expand to the text of the expression, an equals sign, then the representation of the evaluated expression. **This is great for self-documenting and debugging purposes**. Here's an example that shows the difference in behavior:

```
user = 'heinrich'
password = 'super-secret'
f"Log in with: {user} and {password}"
# result: 'Log in with heinrich and super-secret'

f"Log in with: {user=} and {password=}"
# result: "Log in with: user='heinrich' and password='super-secret'"
```

• Tuples:

◇ The last immutable sequence type we are going to look at here is the tuple. **A tuple is sequence of arbitrary Python objects.** In a tuple declaration, items are separated by commas. Tuples are used everywhere in Python. They allow for patterns that are quite hard to reproduce in other languages. Sometimes tuples **are used** implicitly; for example, to set up multiple variables on one line, or **to allow a function to return multiple objects** (in several languages, it is common for a function to return only one object), and in the Python console, tuples can be used implicitly to print multiple elements with one single instruction. We'll see examples for all these cases:

```
t = () # empty tuple
type(t)
# result: <class 'tuple'>

one_element_tuple = (42, ) # you need the comma!
three_elements_tuple = (1, 3, 5) # braces are optional here

a, b, c = 1, 2, 3 # tuple for multiple assignment
a, b, c # implicit tuple to print with one instruction
# result: (1, 2, 3)

3 in three_elements_tuple # membership test
# result: True
```

- Notice that the membership operator can also be used with:

- lists
- strings
- dictionaries
- and in general, with collection and sequence objects.

→ **Note:** Notice that to create a tuple with one item, we need to put a comma after the item. The reason is that without the comma that item is wrapped in braces on its own, in what can be considered a redundant mathematical expression. Notice also that on assignment, braces are optional, so:

- ⇒ **my_tuple = 1, 2, 3**
- is the same as:
- ⇒ **my_tuple = (1, 2, 3).**

◇ One thing that tuple assignment allows us to do is one-line swaps, with no need for a third temporary variable. Let's first see the traditional way of doing it:

```
a, b = 1, 2
c = a # we need three lines and a temporary var c
a = b
b = c
a, b # result: a and b have been swapped
(2, 1)
```

◇ Now let's take a look at the modern way of doing it:

```
a, b = 0, 1
a, b = b, a # this is the Pythonic way to do it
a, b
# result: (1, 0)
```

◇ **Because they are immutable, tuples can be used as keys for dictionaries** (we'll see this shortly). **To us Tuples are Python's built in data that most closely represent a mathematical vector.** This doesn't mean that this was the reason for which they were created, though. **Tuples usually contain a heterogenous sequence of elements** while, on the other hand, **lists are, most of the time, homogenous. Moreover, tuples are normally accessed via unpacking or indexing, while lists are usually iterated over.**

2.5 Mutable Sequences

- **Mutable sequences differ from their immutable counterparts in that they can be changed after creation.** There are two mutable sequence types in Python:

- **lists**
- **byte arrays.**

- **Lists:**

- ◊ **Python lists are very similar to tuples, but they don't have the restrictions of immutability.** Lists are commonly used for storing collections of homogeneous objects , but there is nothing preventing you from storing heterogeneous collections as well.

Lists can be created in many different ways. Let's see an example:

```
[ ] # empty list

list() # same as [ ]
# result: [ ]

[1, 2, 3] # as with tuples, items are comma separated
# result: [1, 2, 3]

[x + 5 for x in [2, 3, 4]] # Python is magic
# result: [7, 8, 9]

list((1, 3, 5, 7, 9)) # list from a tuple
# result: [1, 3, 5, 7, 9]

list('hello') # list from a string
# result: ['h', 'e', 'l', 'l', 'o']
```

- In the previous example, we showed you how to create a list using various techniques.

- We would like you to take a good look at the line with the comment **Python is magic** , which we don't expect you to fully understand at this point. That is called a **list comprehension** : a very powerful functional feature of Python.

- ◊ Creating lists is good, but the real fun begins when we use them, so let's see the main methods they gift us with:

```
a = [1, 2, 1, 3]
a.append(13) # we can append anything at the end
a
# result: [1, 2, 1, 3, 13]

a.count(1) # how many 1's are there in the list?
# result: 2

a.extend([5, 7]) # extend the list by another (or sequence)
a
# result: [1, 2, 1, 3, 13, 5, 7]

a.index(13) # position of '13' in the list (0-based indexing)
# result: 4
```

```

a.insert(0, 17) # insert '17' at position 0
a
# result: [17, 1, 2, 1, 3, 13, 5, 7]

a.pop() # pop (remove and return) last element
# result: 7

a.pop(3) # pop element at position 3
# result: 1
a
# result: [17, 1, 2, 3, 13, 5]

a.remove(17) # remove '17' from the list
a
# result: [1, 2, 3, 13, 5]

a.reverse() # reverse the order of the elements in the list
a
# result: [5, 13, 3, 2, 1]

a.sort() # sort the list
# result: [1, 2, 3, 5, 13]

a.clear() # remove all elements from the list
a
# result: []

```

- The preceding code gives you a roundup of a list's main methods.
- We will see how powerful they are, using the method **extend()** as an example, you can extend lists using any sequence type:

-

```

a = list('hello') # makes a list from a string
a
# result: ['h', 'e', 'l', 'l', 'o']

a.append(100) # append 100, heterogenous type
a
# result: ['h', 'e', 'l', 'l', 'o', 100]

a.extend((1, 2, 3)) # extend using tuple
a
# result: ['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3]

a.extend('...') # extend using string
a
# result: ['h', 'e', 'l', 'l', 'o', 100, 1, 2, 3, '.', '.', '.']

```

◇ Now let's see the most common operations you can do with lists:

```

a = [1, 3, 5, 7]
min(a) # minimum value in the list
# result: 1

max(a) # maximum value in the list
# result: 7

```

▪


```

sum(a) # sum of all values in the list
# result: 16

from math import prod
prod(a) # product of all values in the list
# result: 105

len(a)
# result: 4

b = [6, 7, 8]
a + b # '+' with list means concatenation
# result: [1, 3, 5, 7, 6, 7, 8]

a * 2 # '*' has also a special meaning
# result: [1, 3, 5, 7, 1, 3, 5, 7]

```

- Notice how easily we can perform the **sum** and the **product** all values in a list. The function **prod()**, from the **math** module, is just one of the many new additions introduced in **Python 3.8**. Even if you don't plan to use it that often, it's always a good idea to check out the **math** module and be familiar with its functions, as they can be quite helpful.

- The last two lines in the preceding code are also quite interesting, as they introduce us to a concept called **operator overloading**. In short, this means that operators, such as **+**, **-**, *****, **%**, and so on, may represent different operations according to the context they are used in. It doesn't make any sense to sum two lists, right? Therefore the **+** sign is used to concatenate them. Hence, the ***** sign is used to concatenate the list to itself according to the right operand.

◇ Now, let's take a step further and see something a little more interesting. We want to show you how powerful the **sorted** method can be and how easy it is in Python to achieve results that require a great deal of effort in other languages:

```

from operator import itemgetter
a = [(5, 3), (1, 3), (1, 2), (2, -1), (4, 9)]
sorted(a)
# result: [(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]

sorted(a, key=itemgetter(0))
# result: [(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]

sorted(a, key=itemgetter(0, 1))
# result: [(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]

sorted(a, key=itemgetter(1))
# result: [(2, -1), (1, 2), (5, 3), (1, 3), (4, 9)]

sorted(a, key=itemgetter(1), reverse=True)
# result: [(4, 9), (5, 3), (1, 3), (1, 2), (2, -1)]

```

- In the preceding code, **a** is a list of tuples. This means each element in **a** is a tuple (a 2-tuple in this case).

- When we call **sorted(my_list)**, we get a sorted version **my_list**.

→ **In this case the sorting on a 2-tuple works by sorting them on the first item in the tuple, and on the second when the first one is the same.** You can see this behavior in the result of **sorted(a)**, which yields **[(1, 2), (1, 3), (2, -1), (4, 9), (5, 3)]**.

- Python also gives us the ability to control which element(s) of the tuple the sorting must

be run against.

→ Notice that when we instruct the **sorted** function, to work on the first element of each tuple (with **key=itemgetter(0)**), the result is different: **[(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]**

⇒ The sorting is done only on the first element of each tuple (which is the one at position 0).

→ If we want to replicate the default behavior of a simple **sorted(a)**, we need to use the **key=itemgetter(0, 1)**, which tells Python to sort first on the elements at position **0** within the tuples, and then on those at position **1**. Compare the results and you will they match.

- For completeness. an example of sorting only on the elements at position 1, and then again, with the same sorting but in reverse order.

▪ The Python sorting algorithm is very powerful, and it was written by Tim Peters. It is aptly named, **Timsort**, and it is a blend between **merge** and **insertion sort** and has better time performances than most other algorithms used for mainstream programming languages. Timsort is a stable **sorting algorithm**, which means that when multiple records score the same in the comparison, their original order is preserved. We've seen this in the result of **sorted(a, key=itemgetter(0))**, which yielded **[(1, 3), (1, 2), (2, -1), (4, 9), (5, 3)]**, in which the order of those tuple had been preserved because they had the same value at position **0**.

• Bytearrays:

◇ To conclude our overview of the mutable sequence types, let's spend a moment of the **bytearray** type. Basically, they represent the mutable version of bytes objects. They expose most of the usual methods of mutable sequences as well as most of the methods of the bytes array type. Items in a bytearray are integers in the range **[0, 256)**.

▪ **Note:** When it comes to intervals, we are going to use the standard notation for open/closed ranges. A **square bracket on one end means that the value is included**, while a **round bracket means that it is excluded**. The granularity is usually inferred by the type of the edge elements so, for example, the interval **[3, 7]** means all integers between 3 and 7, inclusive. On the other hand, **(3, 7)** means all integers between 3 and 7, exclusive (4, 5 and 6).

◇ **Items in a bytearray type are integers between 0 and 256; 0 is included, 256 is not;** One reason that intervals are often expressed like this is to ease coding.

◇ Let's see an example with the **bytearray** type:

```
bytearray() #empty bytearray object
# result: bytearray(b'')

bytearray(10) # zero-filled instance with given length
# result: bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')

bytearray(range(5)) # bytearray from iterable of integers
# result: bytearray(b'\x00\x01\x02\x03\x04')

name = bytearray(b'Lina') # A - bytearray from bytes
name.replace(b'L', b'l')
```

```
# result: bytearray(b'lina')
name.endswith(b'na')
# result: True

name.upper()
# result: bytearray(b'LINA')

name.count(b'L')
# result: 1
```

- As you can see, there are a few ways to create a **bytearray** object.

- **They can be useful in many situations; for example, when receiving data through a socket, they eliminate the need to concatenate data while polling, hence they can prove to be very handy.**

- On line **#A**, we created a **bytearray** named as **name** from the bytes literal **b'Lina'** to **show you how the bytearray object exposes methods from both sequences and strings**, which is extremely handy. If you think about it, they can be considered as mutable strings.

2.6 Set Types

- Python also provides two set types:

- ◇ **set**
 - and
- ◇ **frozenset**.

◇ **The set type is mutable, while frozenset is immutable. They are unordered collections of immutable objects.**

◇ **Hashability** is a characteristic that allows an object to be used as a set member as well as a key for a dictionary, as we'll see very soon.

▪ **Note:** from the official Python documentation (<https://docs.python.org/3.9/glossary.html>)
- "An object is hashable if it has a hash value which never changes during its lifetime and can be compared to other objects. **Hashability makes an object usable as a dictionary key and a set member**, because these data structures use the hash value internally. **Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozenset are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default.** They all compare unequal (except with themselves), and their hash value is derived from their **id()** "

◇ Objects that compare equally must have the same value. Sets are very commonly used to test for membership; let's introduce the **in** operator in the following example:

▪

```
small_primes = set() # empty set
small_primes.add(2) # adding one element at a time
small_primes.add(3)
small_primes.add(5)
small_primes
# result: {2, 3, 5}

small_primes.add(1) # look what I've done, 1 is not a prime!
small_primes
#result: {1, 2, 3, 5}

small_primes.remove(1) # so let's remove it
3 in small_primes # membership test
#result: True

4 in small_prime
#result: False

4 not in small_primes # negated membership test
#result: True

small_primes.add(3) # trying to add 3 again
small_primes
#result: {2, 3, 5} # no change, duplication is not allowed

bigger_primes = set([5, 7, 11, 13]) # faster creation
```

```

small_primes | bigger_primes # union operator '|'
#result: {2, 3, 5, 7, 11, 13}

small_primes & bigger_primes # intersection operator '&'
#result: {5}

small_primes - bigger_primes # difference operator '-'
#result: {2, 3}

```

- In the preceding code you can see two different ways to create a set:
- One creates an empty set and then adds elements one at a time.
- The other creates the set **using a list of numbers** as an argument to the constructor, which does all the work for us. Of course, you can create a set from a list or tuple (or any iterable) and then you can add and remove members from the set as you please.

→ **Note:** We'll look at iterable objects and iteration in the next chapter. For now, just know that iterable objects are objects you can iterate on in a direction.

◇ Another way of creating a set is by simply using the curly braces notation, like this:

```

small_primes = {2, 3, 5, 5, 3}
small_primes
# result: {2, 3, 5}

```

- notice we added some duplication to emphasize that the resulting set won't have any.
- Let's see an example using the immutable counterpart of the set type, **frozenset** :

→

```

small_primes = frozenset([2, 3, 5, 7])
bigger_primes = frozenset([5, 7, 11])
small_primes.add(11) # we cannot add to a frozenset
# result: Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'frozenset' object has no attribute 'add'

small_primes.remove(2) # nor can we remove
# result: Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# AttributeError: 'frozenset' object has no attribute 'remove'

small_primes & bigger_primes # intersect, union, etc. is allowed
# result: frozenset({5, 7})

```

⇒ As you can see, **frozenset** objects are quite limited with respect to their mutable counterpart.

- They still prove very effective for **membership test**, **union**, **intersection**, and **difference operations**, and **for performance reasons**.

2.7 Mapping types: Dictionaries

- Of all the built-in data types, the dictionary is easily the most interesting. It's the only standard mapping type, and it is the backbone of every Python object.

- **A dictionary maps keys to values. Keys need to be hashable objects, while values can be of any arbitrary type.**

- ◇ **Dictionaries are also mutable objects.**

- ◇ There are quite a few different ways to create a dictionary, so let us give you a simple example of how to create a dictionary equal to **{'A': 1, 'Z': -1}** in five different ways:

```
a = dict(A=1, Z=-1)
b = {'A': 1, 'Z': -1}
c = dict(zip(['A', 'Z'], [1, -1]))
d = dict([('A', 1), ('Z', -1)])
e = dict({'Z': -1, 'A': 1})

a == b == c == d == e # are they all the same?
# result: True # They are indeed
```

- Have you noticed those double equals? Assignment is done with one equal, while to check whether an object is the same as another one (or five in one go, in this case), we use the double equals.

- There is also another way to compare objects, which involves the **is** operator, and checks whether the two objects are the same (**that is, that they have the same ID, not just the same value**), but unless you have a good reason to use it, you should use the double equals instead. In the preceding code, we also used one nice function: **zip()**. It is named after the real-life zip, which glues together two parts, taking one element from each part at a time. Let's see an example:

→

```
list(zip(['h', 'e', 'l', 'l', 'o'], [1, 2, 3, 4, 5]))
# result: [('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]

list(zip('hello', range(1, 6))) # equivalent, more Pythonic
# result: [('h', 1), ('e', 2), ('l', 3), ('l', 4), ('o', 5)]
```

⇒ In the preceding example, we have created the same list in two different ways, one more explicit, and the other a little bit more Pythonic.

- Forget for a moment that we had to wrap the **list()** constructor around the **zip()** call (the reason is **zip()** returns an iterator, not a **list**, so if we want to see the result, we need to exhaust that iterator into something (a list in this case).), and concentrate on the result. See how **zip()** has coupled the first elements of its two arguments together, then the second ones, then the third ones, and so on?

- ◇ Take a look at the zip of your suitcase, or a purse, or the cover of a pillow, and you will see it works exactly like the one in Python. But let's go back to dictionaries and see how many wonderful methods they expose for allowing us to manipulate them as we want. Let's start with the basic operations:

▪

```

d = {}
d['a'] = 1 # let's set a couple of (key, value) pairs
d['b'] = 2
len(d) # how many pairs
# result: 2

d['a'] # what is the value of 'a'?
# result: 1

d # how does 'd' look now
# result: {'a': 1, 'b': 2}

del d['a'] # let's remove 'a'
d
# result: {'b': 2}

d['c'] = 3 # let's add 'c': 3
'c' in d # membership is checked against the keys
# result: True

3 in d # not the values
# result: False

'e' in d
False

d.clear() # let's clean everything from this dictionary
d
# result: {}

```

- Notice how accessing keys of a dictionary, regardless of the type of operation we're performing, is done using square brackets. Do you remember, strings, lists, and tuples? We were accessing elements at some position through square brackets as well, which is yet another example of Python's consistency.

◇ Let's now take a look at three special objects called dictionary views : **keys**, **values**, and **items**. These objects provide a dynamic view of the dictionary entries and they change when the dictionary changes. **keys()** returns all the keys in the dictionary, **values()** returns all the values in the dictionary, and **items()** returns all the (key, value) pairs in the dictionary . Let's put all this down into code:

```

d = dict(zip('hello', range(5)))
d
# result: {'h': 0, 'e': 1, 'l': 3, 'o': 4}

d.keys()
# result: dict_keys(['h', 'e', 'l', 'o'])

d.values()
# result: dict_values([0, 1, 3, 4])

d.items()
# result: dict_items([('h', 0), ('e', 1), ('l', 3), ('o', 4)])

3 in d.values()
# result: True

```

```
('o', 4) in d.items()  
# result: True
```

- There are a few things to note here.

→ First, notice how we are creating a dictionary by iterating over the zipped version of the string **'hello'** and the list **[0, 1, 2, 3, 4]**.

→ The string **'hello'** has two **'l'**

' characters inside, and they are paired up with the values **2** and **3** by the **zip()** function. Notice how in the dictionary, the second occurrence of the **'l'** key (**the one with value 3**), overwrites the first one (**the one with the value 2**).

→ Another thing to notice is that when asking for any view, the original order in which items were added is now preserved, while before version **3.6** there was no guarantee of that.

◇ As of Python 3.6, the **dict** type has been re-implemented to use a more compact representation. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5. Moreover, since Python 3.6, as a side effect, dictionaries preserve the order in which keys were inserted. This feature has received such a welcome from the community that in 3.7 it has become an official feature of the language rather than an implementation side effect. Since Python 3.8, dictionaries are also reversible.

◇ We'll see how these views are fundamental tools when we talk about iterating over collections. Let's take a look now at some other methods exposed by Python's dictionaries (there's plenty of them and they're very useful):

```
d  
# result: {'h': 0, 'e': 1, 'l': 3, 'o': 4}  
  
d.popitem()  
# result: ('o', 4)  
  
d.pop('l') # remove item with key 'l'  
# result: 3  
  
d.pop('not-a-key') # remove a key not in dictionary  
# result: Traceback (most recent call last):  
#   File "<stdin>", line 1, in <module>  
# KeyError: 'not-a-key'  
  
d.pop('not-a-key', 'default-value') # with a default value?  
# result: 'default-value'  
  
d.update({'another': 'value'}) # we can update dict this way  
d  
# result: {'h': 0, 'e': 1, 'another': 'value'}  
  
d.update(a=13) # or this way (like a function call)  
d  
# result: {'h': 0, 'e': 1, 'another': 'value', 'a': 13}  
  
d.get('a') # same as d['a'] but if key is missing no KeyError  
# result: 13  
  
d.get('a', 177) # default value used if key is missing
```



```
# result: 13
d.get('b', 177) # like in this case
# result: 177

d.get('b') # key is not there, so None is returned
```

- All these methods are quite simple to understand, but it's worth talking about that **None**, for a moment.

- Every function in Python returns **None**, unless the **return** statement is explicitly used to return something else, but we'll see this when we explore functions.

→ **None** is frequently used to represent the absence of a value, and it is quite commonly used as a default value for arguments in function declaration.

⇒ Some inexperienced coders sometimes write code that returns either **False** or **None**. both **False** and **None** evaluate to **False** in Boolean context, so it may seem that there is not much difference between them. But actually, we would argue the contrary, that there is an important difference: **False means that we have information**, and the information we have is **False**. **None means no information**; **no information** is very different from information that is **False**. In layman's terms, if you ask your mechanic Is my car ready?, there is a big difference between the answer No, it's not (**False**) and I have no idea (**None**).

◇ One last method we really like about dictionaries is **setdefault()**. It behaves like **get()**, but also sets the key with the given value if it is not there. Let's see an example:

```
d = {}
d.setdefault('a', 1) # 'a' is missing, we get default value
# result: 1

d
# result: {'a': 1} # also the key/value pair ('a', 1) has now been added

d.setdefault('a', 5) # let's try to override the value
# result: 1

d
# result: {'a': 1} # no override as expected
```

◇ This brings us to the end of this tour of dictionaries. Test your knowledge about them by trying to foresee what **d** looks like after this line:

```
d = {}
d.setdefault('a', {}).setdefault('b', []).append(1)
```

◇ Python 3.9 sports a brand-new union operator available for **dict** objects, which was introduced by PEP 584.

▪ When it comes to applying union to **dict** objects, we need to remember that union for them is not commutative. This becomes evident when the two **dict** objects we're merging have one or more keys in common. Check out this example:

```
- d = {'a': 'A', 'b': 'B'}
```

```
e = {'b': '8', 'c': 'C'}
d | e
# result: {'a': 'A', 'b': 8, 'c': 'C'}

e | d
# result: {'b': 'B', 'c': 'C', 'a': 'A'}

{**d, **e}
# result: {'a': 'A', 'b': '8', 'c': 'C'}
```

→ Here, **dict** objects **d** and **e** have the key '**b**' in common.

→ For the **dict** object, **d**, the value associated with '**b**' is '**B**'; whereas, for **dict e**, it's the number **8**.

⇒ This means that when we merge them with **e** on the right hand side of the union operator, **|**, the value in **e** overrides the one in **d**.

⇒ The opposite happens, of course, when we swap the positions of those objects in relation to the union operator.

→ In this example, you can also see how the **union** can be performed by using the ****** operator to produce a **dictionary unpacking**.

→ It's worth noting that union can also be performed as an augmented assignment operation (**d |= e**), which works in place. Please refer to **PEP 584** for more information about this feature.

2.8 Data Types

- Python provides a variety of specialized data types, such as **dates** and **times**, **container types**, and **enumerations**. There is a whole section in the Python standard library titled Data Types, which deserved to be explored; it is filled with interesting and useful tools for each and every programmer's needs.

- You can find it here → <https://docs.python.org/3/library/datatypes.html>

- In this section, we are briefly going to take a look at:

- ◇ **dates** and **times**

- ◇ **collections**

- ◇ **enumerations**.

- **Dates and times:**

The Python standard library provides several data types that can be used to deal with dates and times. this realm may seem innocuous at first glance, but it's actually quite tricky: time zones, daylight saving

time... There are a huge number of ways to format date and time information; calendar quirks, parsing, and localizing; these are just a few of the many difficulties we face when we deal with dates and times, and

that's probably the reason why, in this particular context, it is very common for Python professional programmers to also rely on various third-party libraries that provide some much-needed extra power.

- ◇ **The standard library:**

- We will start with the **standard library**, and finish the session with a little overview of what's out there in terms of the third-party libraries you can use.

- From the standard library, the main modules that are used to handle dates and times are **datetime**, **calendar**, **zoneinfo**, and **time**. Let's start with the imports you'll need for this whole section:

-

```
from datetime import date, datetime, timedelta, timezone
import time
import calendar as cal
from zoneinfo import ZoneInfo
```

- The first example deals with dates. Let's see how they look:

-

```
today = date.today()
today
# result: datetime.date(2022, 12, 19)

today.ctime()
# result: 'Mon Dec 19 00:00:00 2022'

today.isoformat()
# result: '2022-12-19'
```

```

today.weekday()
# result: 0

cal.day_name[today.weekday()]
# result: 'Monday'

today.day, today.month, today.year
# result: (19, 12, 2022)

today.timetuple()
# result: time.struct_time(tm_year=2022, tm_mon=12, tm_mday=19, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=0, tm_yday=353, tm_isdst=-1)

```

→ We start by fetching the date for today. We can see that it's an instance of the **datetime.date** class. Then we get two different representations for it, following the C and ISO 8601 format standards, respectively. After that, we ask what day of the week it is, and we get the number 6. Days are numbered 0 to 6 (representing Monday to Sunday), so we grab the value of the sixth element in `calendar.day_name` (notice in the code that we have substituted **calendar** with "**cal**" for brevity).

→ The last two instructions show how to get detailed information out of a date object. We can inspect its **day**, **month** and **year** attributes, or call the **timetuple()** method and get a whole wealth of information. Since we're dealing with a date object, notice that all the information about time has been set to 0.

- Let's now play with time:

```

time.ctime()
# result: 'Mon Dec 19 15:24:49 2022'

time.daylight
# result: 0

time.gmtime()
# result: time.struct_time(tm_year=2022, tm_mon=12, tm_mday=19, tm_hour=20,
tm_min=27, tm_sec=6, tm_wday=0, tm_yday=353, tm_isdst=0)

time.gmtime(0)
# result: time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)

time.localtime()
# result: time.struct_time(tm_year=2022, tm_mon=12, tm_mday=19, tm_hour=15,
tm_min=28, tm_sec=6, tm_wday=0, tm_yday=353, tm_isdst=0)

time.time()
# result: 1671481745.3147464

```

→ This example is quite similar to the one before, only here, we are dealing with time. We can see how to get a printed representation of time **according to C format standard**, and **then how to check if daylight saving time is in effect**. The function **gmtime** converts a **given number of seconds from the epoch to a struct_time object in UTC**. If we don't feed it any number, it will use the current time.

→ We finish the example by getting the object for the current local time and the number

of seconds from the epoch expressed as a float number:

⇒ **time.time()**

▪ Let's now see an example using **datetime** object, **which brings together dates and times**:

-

```
now = datetime.now()
utcnow = datetime.utcnow()

now
# result: datetime.datetime(2022, 12, 19, 15, 39, 47, 433418)
utcnow
# result: datetime.datetime(2022, 12, 19, 20, 40, 42, 504282)

now.date()
# result: datetime.date(2022, 12, 19)

now.day, now.month, now.year
# result: (19, 12, 2022)

now.date() == date.today()
# result: true

now.time()
#result: datetime.time(15, 40, 34, 360165)

now.hour, now.minute, now.second, now.microsecond
#result:(15, 40, 34, 360165)

now.ctime()
#result: 'Mon Dec 19 15:40:34 2022'

now.isoformat()
#result: '2022-12-19T15:40:34.360165'

now.timetuple()
#result: time.struct_time(tm_year=2022, tm_mon=12, tm_mday=19, tm_hour=15,
tm_min=40, tm_sec=34, tm_wday=0, tm_yday=353, tm_isdst=-1)

now.tzinfo
utcnow.tzinfo
now.weekday()
#result: 0
```

→ The preceding example is rather self explanatory:

⇒ We start by setting up two instances that represent the current time. One is related to UTC (**utcnow**), and the other one is a local representation (**now**).

• You can get **date**, **time**, and specific attributes from a **datetime** object in a similar way as to what we have already seen. It is also worth noting how both **now** and **utcnow** present the value **None** for the **tzinfo** attribute. This happens because those objects are **naive**.

▪ **Note:** **Date** and **time** objects **can be categorized as aware if they include time zone information**, or **naive** if they don't.

◇ Let's now see how a duration is represented in this context:

```

f_bday = datetime(1975, 12, 29, 12, 50, tzinfo=ZoneInfo('Europe/Rome'))

h_bday = datetime(
    1981, 10, 7, 15, 30, 50, tzinfo=timezone(timedelta(hours=2))
)

diff = h_bday - f_bday
type(diff)
# result: <class 'datetime.timedelta'>

diff.days
# result: 2109

diff.total_seconds()
# result: 182223650.0

today + timedelta(days=49)
# result: datetime.date(2021, 5, 16)

now + timedelta(weeks=7)
# result: datetime.datetime(2021, 5, 16, 25, 16, 258274)

```

- Two objects have been created that represent Fabrizio and Heinrich's birthdays. This time, in order to show you the alternative, we have create **aware** objects.

- There are several ways to include time zone information when creating a **datetime** object, and in this example, we are showing you two of them. one uses the brand-new **ZoneInfo** object from the **zoneinfo** module, introduced in Python 3.9. the second uses a simple **timedelta**, an object that represents a duration.

→ **page 104**

- We then create the **diff** object, which is assigned as the subtraction of them. The result of that operation is an instance of **timedelta**. You can see how we interrogate the **diff** object to tell us how many days Fabrizio and Heinrich's birthday are apart, and even the number of seconds that represent that whole duration. Notice that we need to use **total_seconds**, which expresses the whole duration in seconds. The **seconds** attribute represents the number of seconds assigned to that duration. So a **timedelta(days=1)** will have seconds equal to **0**, and **total_seconds** equal to **86,400** (which is the number of seconds in a day).

- Combining a **datetime** with a duration adds or subtracts that duration from the original date and time information. In the last few lines of the example, we can see how adding a duration to a date object produces a **date** as a result, whereas adding it to a **datetime**, as it is fair to expect.

◇ One of the more difficult undertakings to carry out using dates and times is parsing. Let's see a short example:

```

datetime.fromisoformat('1977-11-24T19:30:13+01:00')
# result: datetime.datetime(1977, 11, 24, 19, 30, 13,
tzinfo=datetime.timezone(datetime.timedelta(seconds=3600)))

```

```
datetime.fromtimestamp(time.time())  
# result: datetime.datetime(2021, 3, 28, 15, 42, 2, 142696)
```

- We can easily create **datetime** objects from ISO-formatted strings, as well as from timestamps. However, in general, parsing a date from unknown formats can prove to be a difficult task.

2.8.1 Third-party libraries

- To finish off this subsection, we would like to mention a few third-party libraries that you will very likely come across the moment you will have to deal with dates and times in your code:

- ◇ **dateutil:**

- Powerful extension to **datetime**
 - <https://dateutil.readthedocs.io/en/stable/>

- ◇ **Arrow:**

- Better date and time for Python
 - <https://arrow.readthedocs.io/en/latest/>

- ◇ **pytz:**

- World time zone definitions for Python
 - <https://pythonhosted.org/pytz/>

- These three are some of the most common, and they are worth investigating.
- Let's take a look at one final example, this time using the **Arrow** third-party library:

◇

```
import arrow
arrow.utcnow()
# result: <Arrow [2022-12-20T13:46:42.122384+00:00]>

arrow.now()
# result: <Arrow [2022-12-20T08:48:42.621783-05:00]>

local = arrow.now('Europe/Rome')
local
# result: <Arrow [2022-12-20T14:49:50.028696+01:00]>

local.to('utc')
# result: <Arrow [2022-12-20T13:50:09.724853+00:00]>

local.to('Europe/Moscow')
# result: <Arrow [2022-12-20T16:50:09.724853+03:00]>

local.to('Asia/Tokyo')
# result: <Arrow [2022-12-20T22:50:09.724853+09:00]>

local.datetime
# result datetime.datetime(2022, 12, 20, 14, 50, 9, 724853,
tzinfo=tzfile('Arctic/Longyearbyen'))

local.isoformat()
# result '2022-12-20T14:50:09.724853+01:00'
```

- Arrow provides a wrapper around the data structures of the standard library, plus a whole set of methods and helpers that simplify the task of dealing with dates and times. You can see from this example how easy it is to get the local date and time in the Italian time zone (Europe/Rome), as well as to convert it to UTC, or to the Russian or Japanese time zones. The last two

instructions show how you can get the underlying **datetime** object from an Arrow one, and the very useful ISO-formatted representation of a date and time.

The collections module

- When Python general-purpose built-in containers (**tuple**, **list**, **set** and **dict**) aren't enough, we can find specialized container data types in the **collections** module:



Data type	Description
<code>namedtuple()</code>	Factory function for creating tuple subclasses with named fields
<code>deque</code>	List-like container with fast appends and pops on either end
<code>ChainMap</code>	Dictionary-like class for creating a single view of multiple mappings
<code>Counter</code>	Dictionary subclass for counting hashable objects
<code>OrderedDict</code>	Dictionary subclass with methods that allow for re-ordering entries
<code>defaultdict</code>	Dictionary subclass that calls a factory function to supply missing values
<code>UserDict</code>	Wrapper around dictionary objects for easier dictionary subclassing
<code>UserList</code>	Wrapper around list objects for easier list subclassing
<code>UserString</code>	Wrapper around string objects for easier string subclassing

- There isn't enough space here to cover them all, but you can find plenty of example in the official documentation; here, we will just give a small example to show you **namedtuple**, **defaultdict**, and **ChainMap**.

◇ **namedtuple**:

- It's a tuple-like object that has fields accessible by attribute lookup, as well as being **indexable** and **iterable** (it's actually a subclass of **tuple**). This is sort of a compromise between a fully-fledged object and a tuple, and it can be useful in those case where you don't need the full power of a custom object, but only want your code to be more readable by avoiding weird

indexing. Another use case is when there is a chance that items in the tuple need to change their position after refactoring, forcing the coder to also refactor all the logic involved, which can be very tricky.

- For example, say we are handling data about the left and right eyes of a patient. We save one value for the left eye (position 0) and one for the right eye (position 1) in a regular tuple. Here's how that may look:

```
vision = (9.5, 8.8)
vision
# result: (9.5, 8.8)

vision[0]
# result: 9.5

vision[1]
# result: 8.8
```

→ Now let's pretend we handle vision objects all of the time, and, at some point, the designer needs to enhance them by adding information for the combined vision, so that a vision object stores data in this format (left eye, combined, right eye).

→ Do you see the trouble we're in now? We may have a lot of code that depends on **vision[0]** being the left eye information (which it still is) and **vision[1]** being the right eye information (which is no longer the case). We have to refactor our code wherever we handle these objects, changing **vision[1]** to **vision[2]**, and that can be painful. We could have probably approached this a bit better from the beginning, by using a **namedtuple**. Let us show you what we mean:

⇒

```
from collections import namedtuple
Vision = namedtuple('Vision', ['left', 'right'])
vision = Vision(9.5, 8.8)
vision[0]
# result: 9.5

vision.left # same as vision[0], but explicit
# result: 9.5

vision.right # same as vision[1], but explicit
# result: 8.8
```

- If, within our code, we refer to the left and right and eyes using **vision.left** and **vision.right**, all we need to do to fix the new design issue is change our factory and the way we create instances (the rest of the code won't need to change):

◇

```
Vision = namedtuple('Vision', ['left', 'combined', 'right'])
vision = Vision(9.5, 9.2, 8.8)
vision.left # still correct
# result: 9.5

vision.right # still correct
# result: 8.8

vision.combined # still correct (though now is vision[2])
```

```
# result: 8.8
vision.combined # the new vision[1]
# result: 9.2
```

▪ You can see how convenient it is to refer to those values by name rather than by position. After all, as a wise man once wrote, Explicit is better than implicit. This example may be a little extreme; of course, it's not likely that our code designer will go for a change like this, but you'd be amazed to see how frequently issues similar to this one occur in a professional environment, and how painful it is to refactor in such cases.

◇ **defaultdict:**

▪ The **defaultdict** data type is one of the authors' favorites. It allows you to avoid checking whether a key is in a dictionary by simply inserting it for you on your first access, with a default value whose type you pass on creation. In some cases, this tool can be very handy and shorten your code a little. Let's see a quick example. Say we are updating the value of **age**, by adding one year. if **age** is not there, we assume it was and we update it to **1**:

```
d = {}
d['age'] = d.get('age', 0) + 1
d
# result: {'age': 1}
d = {'age': 39}
d['age'] = d.get('age', 0) + 1
d
# result: {'age': 40}
```

▪ Now let's see how it would work with the **defaultdict** data type. The second line is actually the short version of an **if** clause that runs to a length of four lines, and that we would have to write if dictionaries didn't have the **get()** method:

```
from collections import defaultdict
dd = defaultdict(int) # int is the default typw (0 the value)
dd['age'] += 1 # short for dd['age'] = dd['age'] + 1
dd
# result: defaultdict(<class 'int'>, {'age': 1}) # 1, as expected
```

→ Notice how we just need to instruct the **defaultdict** factory that we want an **int** number to be use if the key is missing (we'll get **0**, which is the default for the **int** type). Also notice that even though in this example there is no gain on the number of lines, there is definitely a gain in readability, which is very important. You can also use a different technique to instantiate a **defaultdict** data type, which involves creating a factory object.

◇ **ChainMap:**

▪ It's an extremely useful data type which was introduced in Python 3.3. It behaves like a normal dictionary but, according to Python documentation, is provided for quickly linking a number of mappings so they can be treated as a single unit. This is usually much faster than creating one dictionary and running multiple **update** calls on it. **ChainMap** can be used to simulate nested scopes and is useful in templating. The underlying mappings are stored in a list.

That list is public and can be accessed or updated using the **maps** attribute. Lookups search the underlying mappings successively until a key is found. By contrast, writes, updates, and deletions only operate on the first mapping.

- A very common use case is providing defaults, so let's see an example:

```
default_connection = {'host': 'localhost', 'port': 4567}
connection = {'port': 5678}
conn = ChainMap(connection, default_connection) # map creation
conn['port'] # port is found in the first dictionary
# result: 5678

conn['host'] # host is fetched from the second dictionary
# result: localhost

conn.maps # we can see the mapping objects
# result: [{'port': 5678}, {'host': 'localhost', 'port': 4567}]

conn['host'] = 'packtpub.com' # let's add host
conn.maps
# result: ['port': 5678, 'host': 'packtpub.com'], {'host': 'localhost', 'port': 4567}]

conn['port'] # now port is fetched from the second dictionary
# result: 4567

dict(conn) # easy to merge and convert to regular dictionary
# result: {'host': 'packtpub.com', 'port': 4567}
```

2.8.2 Enums

- Technically not a built-in data type, as you have to import them from the **enum** module, but definitely worth mentioning, are **enumerations**. They were introduced in Python 3.4, and though it is not that common to see them in professional code, we thought it would be a good idea to give you an example anyway.

- The official definition of an enumeration is that it is:

- ◇ A set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over.

- Say that you need to represent traffic lights; in your code, you might resort to the following:

```
GREEN = 1
YELLOW = 2
RED = 4
TRAFFIC_LIGHTS = (GREEN, YELLOW, RED)

# or with a dict
traffic_lights = {'GREEN': 1, 'YELLOW': 2, 'RED': 4}
```

→ There's nothing special about this code. It's something, in fact, that is very common to find. But, consider doing this instead:

⇒

```
from enum import Enum
class TrafficLight(Enum):
    GREEN = 1
    YELLOW = 2
    RED = 4

TrafficLight.GREEN
# result: <TrafficLight.GREEN: 1>

TrafficLight.GREEN.name
# result: 'GREEN'

TrafficLight.GREEN.value
# result: 1

TrafficLight(1)
# result: <TrafficLight.GREEN: 1>

TrafficLight(4)
# result: <TrafficLight.RED: 4>
```

- Ignoring for a moment the (relative) complexity of a class definition, you can appreciate how this approach may be advantageous. The data structure is much cleaner, and the API it provides is much more powerful. We encourage you to check out the official documentation to explore all the great features you can find in the **enum** module. We think it's worth exploring at least once.

2.8.3 Final considerations

- That's it. Now you have seen a very good proportion of the data structures that you will use in Python.
- Before we leap into Chapter 3, we're gonna see some final considerations about different aspects that, to our minds, are important not to be neglected.

2.8.3.1 Small value caching

- **Small value caching:**

◇ While discussing objects at the beginning of this chapter, we saw that when we assigned a name to an object, Python create the object, sets its value, and then points the name to it. We can assign different names to the same value, and we expect different objects to be created, like this:

```
a = 1000000
b = 1000000
id(a) == id(b)
# result: False
```

- In the preceding example, **a** and **b** are assigned two **int** objects, which have the same value, but they are not the same object (as you can see, their **id** is not the same). So, let's do it again:

→

```
a = 5
b = 5
id(a) == id(b)
# result: True
```

⇒ Uh-oh! Is Python broken? Why are the two objects the same now? We didn't do **a = b = 5**; we set them up separately.

- Well, the answer is **performance**. Python caches short strings and small numbers to avoid having many copies of them clogging up the system memory. In the case of strings, caching or, more appropriately, interning them, also provides a significant performance improvement for comparison operations. Everything is handled properly under the hood, so you don't need to worry, but make sure that you remember this behavior should your code ever need to fiddle with IDs.

2.8.3.2 How to choose data structures

• How to choose data structures:

◇ As we've seen, Python provides you with several built-in data types and, sometimes, if you're not that experienced, choosing the one that serves you the best can be tricky, especially when it comes to collections. For example, say you have many dictionaries to store, each of which represents a customer. Within each customer dictionary, there's an **'id': 'code'** unique identification code. In what kind of collection would you place them? Well, unless we know more about these customers, it's very hard to answer.

- What kind of access will we need?
- What sort of operations will we have to perform on each of them, and how many time?
- Will the collection change over time?
- Will we need to modify the customer dictionaries in any way?
- What is going to be the most frequent operation we will have to perform on the collection.

◇ If you can answer the preceding questions, then you will know what to choose. If the collection never shrinks or grows (in other words, it won't need to add/delete any customer object after creation) or shuffles, then tuples are a possible choice. Otherwise, lists are a good candidate. Every customer dictionary has a unique identifier though, so even a dictionary could work. Let us draft these options for you:

▪

```
# example customer objects
customer1 = {'id': 'abc123', 'full_name': 'Master Yoda'}
customer2 = {'id': 'def456', 'full_name': 'Obi-Wan Kenobi'}
customer3 = {'id': 'ghi789', 'full_name': 'Anakin Skywalker'}

# collects them in a tuple
customers = (customer1, customer2, customer3)

# or collect them in a list
customers = [customer1, customer2, customer3]

# or maybe within a dictionary, they have a unique id after all
customers = {
    'abc123': customer1,
    'def456': customer2,
    'ghi789': customer3,
}
```

- Some customers we have there right? We probably wouldn't go with the tuple option, unless we wanted to highlight that the collection is not going to change. We would say that, usually, a list is better, as it allows for more flexibility.

- Another factor to keep in mind is that tuple and lists are ordered collections. If you use a dictionary (prior to Python 3.6) or a set, you would lose the ordering, so you need to know if ordering is important in your application.

- What about performance? For example, in a **list**, operations such as insertion and membership testing can take $O(n)$ time, while they are $O(1)$ for a **dictionary**. It's not always possible to use dictionaries though, if we don't have the guarantee that we can uniquely identify

each item of the collection by means of one of its properties, and that the property in question is hashable (so it can be a key in **dict**).

→ <https://www.bigocheatsheet.com/>

- Another way of understanding whether you have chosen the right data structure is by looking at the code you have to write in order to manipulate it. If everything comes easily and flows naturally, then you probably have chosen correctly, but if you find yourself thinking your code is unnecessarily complicated, then you probably should try to decide whether you need to reconsider your choices. It's quite hard to give advice without a practical case though, so when you choose a data structure for your data, try to keep ease of use and performance in mind, and give precedence to what matters the most in the context you are in.

2.8.3.3 About indexing and slicing

- At the beginning of this chapter, we saw slicing applied to strings. Slicing, in general applies to a sequence: tuples, lists, strings and so on. With lists, slicing can also be used for assignment. We have almost never seen this used in professional code, but still, you know you can. Could you slice dictionaries or sets? Of course not! Excellent, so let's talk about indexing.
- There is one characteristic regarding Python indexing that we haven't mentioned before. We'll show you by way of an example. How do you address the last element of a collection? Let's see:

```
a = list(range(10)) # 'a' has 10 elements. Last one is 9.
a
# result: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

len(a) # its length is 10 elements
# result: 10

a[len(a) - 1] # position of last one is len(a) - 1
# result: 9

a[-1] # but we don't need len(a)!
# result: 9

a[-2] # equivalent to len(a) - 2
# result: 8

a[-3] # equivalent to len(a) - 3
# result: 7
```

- In order to fetch the last element, we need to know the whole length of the list (or tuple, or string, and so on) and then subtract 1. Hence: **len(a) - 1**. this is so common an operation that Python provides you with a way to retrieve elements using **negative indexing**. this proves very useful when performing data manipulation. The following image shows how indexing works on the string **"HelloThere"**:



2.8.3.4 About names

- You may have noticed that, in order to keep the examples as short as possible, we have names many objects using simple letters, like **a**, **b**, **c**, and so on. This is perfectly fine when debugging on the console or showing that **a + b == 7**, but it's bad practice when it comes to professional coding (or any type of coding, for that matter). We hope you will indulge us where we have done it; the reason is to present the code in a more compact way.
- In a real environment though, when you choose names for your data, you should choose them carefully (they should reflect what the data is about). So if you have a collection of **Customer** objects, **customers** is a perfectly good name for it. Would **customers_list**, **customers_tuple**, or **customers_collection** work as well? Think about it for a second. Is it good to tie the name of the collection to the datatype. We don't think so, at least in most cases. So, if you have an excellent reason to do so, go ahead; otherwise, don't. The reasoning behind this is that once **customers_tuple** starts being used in different places of your code, and you realize you actually want to use a list instead of a tuple, you're up for some fun refactoring (also known as wasted time). **Names for data should be nouns, and names for functions should be verbs.** Names should be as expressive as possible. Python is actually a very good example when it comes to names. Most of the time you can just guess what a function is called if you know what it does.
- Chapter 2 from the book Clean Code by Robert C. Martin is entirely dedicated to names. It's an amazing book that helped us to improve our coding style in many different ways; it is a must-read if you want to take your coding to the next level.

2.9 Summary

- We encourage you to Play with all data types. Exercise them, break them, discover all their methods, enjoy them, and learn them very, very well. If your foundation is not rock solid, how good can your code be? Data is the foundation for everything; data shapes what dances around it.