Refresher on Python

Introduction to Artificial Intelligence (TNM119)

Elmira Zohrevandi

Spring 2025

Linköping University, Norrköping, Sweden

Contents

How to read this refresher?

This refresher is by no means a complete guide or tutorial to the Python programming language. Instead, It is meant to be used as a quick start guide with lots of short examples. It provides a concise overview of the language features and tools tailored to our requirements and use cases in the **TNM119** course. As a result, we might skip some aspects or delve into others more.

In particular, it has been crafted to allow for reading through the entire content quickly. You just need to follow down the examples one by one. Often, the examples build upon the previous ones. In such cases ensure that you also read the preceding examples. You will best benefit from the examples if you use a Jupyter notebook. One does not need any prior knowledge of Python. However, it is assumed you have experience in at least one other programming language such as C, C++, or Java, and are familiar with object-oriented programming.

If you have practical experience with Python programming, then you can perhaps skip directly to the exercises of **Lab Zero**! However, I recommend that you still scan through the pages. It will not take much time anyways and might even save your time later, by informing you about the best practices and *subtleties* of the language!

The list of symbols used in this refresher and how you should interpret them:

 General notes

 Programming tips

 Subtleties of the language

 Python code that needs to be run/executed

 Output of the executed code when successful

 Exceptions (errors) raised as a result of running incorrect code

The brief guide to the different language elements used in this refresher:

`# interesting`	a single-line comment
`variable_one`	a variable (snake_case)
`simple_function()`	a function (snake_case)
`MyClass`	a class (CamelCase)
`.my_method()`	a method of a class, i.e. it should be called on the instance
`import numpy as np`	importing the NumPy library and setting `np` as its alias
`<placeholder>`	a "stand-in" that needs to be replaced by you, e.g. `<filename>` with `"data.csv"`

1 What is Python?

Python is a high-level and general purpose programming language designed by Guido van Rossum in 1991. Owing to its wide adoption by the community and its large eco-system of libraries and tools, Python applications and use cases are versatile. Among many the most notable ones are

- artificial intelligence
- data science
- (interactive) user interfaces and visualizations
- computer graphics
- robotics
- dynamic and feature-rich websites
- games
- phone applications
- computer simulations
- general scripting

Python is both dynamically type-checked (Section 2.1.4) and garbage-collected. It is also an interpreted language, which allows for rapid prototyping with minimal effort. In recent years, Python has been considered as the *de facto* language for artificial intelligence and data science.

There are different implementations for the Python[1] language, e.g. CPython and PyPy. We use **CPython**, since it is the reference implementation.

The following is an example of a recursive implementation of the **fibonacci** function in Python. It includes optional *type hints* (Section 2.1.5) in the signature of the function (Chapter 8),

```
def fibonacci(n: int) -> int:
    """Calculate the nth fibonacci number, where `n` must be a non-negative integer."""

    if not isinstance(n, int):
        raise ValueError("Input must be an integer!")

    if n < 0:
        raise ValueError("Input cannot be negative.")

    if n >= 2:
        return fibonacci(n - 1) + fibonacci(n - 2)
    else:
        return n
```

The following is an example of how to run the function for a number of inputs

```
for i in range(20):
    print(fibonacci(i), end=" ")
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

[1] This is the implementation of the language itself, and not the implementation of a specific program!

2 Types and objects

2.1 Types

An important aspect of any programming language is its type system and built-in types. Python is no exception. Providing a comprehensive explanation of all the python types is beyond the scope of this refresher. Instead, we will only focus on the few ones that we will be frequently using in the course, e.g. numerics (Chapter 3) and collections (Chapters 4 to 7).

> **Note**
>
> If you are interested to know more, refer to
> - Python Language Reference
> - Python objects, values, and types

2.1.1 Built-in `type()` class

> **Tip**
>
> One can get the type of an object using the Python built-in `type()` class.

```
# type of an integer literal
type(20)
```

```
int
```

```
x = 20

# type of a variable whose value is an integer
type(x)
```

```
int
```

```
# type of a string literal
type("hello")
```

```
str
```

2.1.2 Everything is an object

> **Note**
>
> A class is a bundle of methods (functions) and data that are closely related. In the language of object-oriented programming, this is referred to as **encapsulation**.

> **Note**
>
> Classes are **blueprints** for objects, and objects are **instances** of classes. An actual object comes into existence when the class is instantiated. This is similar to having a house as an actual physical object in the real world, which is made out of e.g. concrete, and the blueprint of the same house which exists on a piece of paper.

> **Tip**
>
> Functions, classes, modules, and even the types are also objects in Python. As a result, `type()` works on them as well.

```
# `print` is a built-in function.
type(print)
```

```
builtin_function_or_method
```

```
# `math` is a built-in module and part of the Python standard library.
import math

type(math)
```

```
module
```

```
# `int` is a built-in type, so its type is also `type`!
type(int)
```

```
type
```

> **Tip**
>
> You can use python built-in function `isinstance()` to check whether an object is an instance of a specific class.

> **Tip**
>
> In Python, `object` is the ultimate base class of everything. All other classes inherit from `object`. As a result, everything is an instance of the `object` class.

```
# `print()` function is an instance of the `object` class.
isinstance(print, object)
```

```
True
```

```
# `math` is a module and also an object.
import math

isinstance(math, object)
```

```
True
```

```
# `int` is a type but also an object.
isinstance(int, object)
```

```
True
```

```
# Even `object` itself is an object!
isinstance(object, object)
```

```
True
```

```
# `type` is also an object!
isinstance(type, object)
```

```
True
```

> **Note**
>
> Can you explain why the `type` of `type` is `type`?

```
type(type)
```

```
type
```

> **Note**
>
> You can use the built-in function `print()` to print the value of an expression to the standard output.

Instead of just evaluating the type of something, you can also explicitly print it. Look at the two examples below.

```
x = 1.0
type(x)
```

```
float
```

```
x = 1.0
print(type(x))
```

```
<class 'float'>
```

Printing the type explicitly, reveals the fact the `float` is a class and `x` is an object of this class.

2.1.3 Frequently-used built-in types

Tip

Below is the list of frequently-used built-in types in Python. All of the following are classes, although they resemble functions,

- bool
- dict
- float
- int
- list
- object
- range
- set
- str
- tuple
- type

2.1.4 Dynamic typing

Python is a dynamically type-checked language. This means the type of an object is determined and verified at **runtime**. This provides a flexibility which is advantageous for rapid prototyping.

```
x = 1               # `x` is of type `int`.
print(x, type(x))

x = "hello"         # `x` changed both its value and type. Python is totally fine with it!
print(x, type(x))
```

```
1 <class 'int'>
hello <class 'str'>
```

Due to the dynamic typing, functions can behave differently based on the type of the provided input. For example, let's define a simple function which doubles its input and returns the result,

```
def double(x):
    return x * 2        # What could be the type of `x` in here?
```

```
print(double(3.5))      # It works for floats.
```

```
7.0
```

```
print(double("echo!"))  # It also works for strings as well, interesting!
```

```
echo!echo!
```

However, this can also lead to ambiguity and even failure at runtime. In large and complex code bases, we might deal with input data over which we have marginal or no control at all. This can cause issues and bugs. Let's run our `double()` function again, but this time for a list and a set with the same elements.

```
double([2,3])  # It works for lists, good!
```

```
[2, 3, 2, 3]
```

```
double({2,3})  # Oops! This doesn't work for sets. The server is down! :(
```

```
TypeError: unsupported operand type(s) for *: 'set' and 'int'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 double(2,3)  # Oops! This doesn't work for sets. The server is down! :(
Cell In[15], line 2, in double(x)
      1 def double(x):
----> 2     return x * 2
TypeError: unsupported operand type(s) for *: 'set' and 'int'
```

If only we could **check** for such errors before running the code in production? The next section provides a solution!

2.1.5 Type hints

Python has **type hints**. These are **optional annotations** which add type information to the objects. Using type hints, a program called a **type checker** can scan the code and warn us about such issues prior to running the code in production. In particular, the type checker can be set to stop us from running the program until the type errors are resolved.

There are a number of type checkers for Python that you can install. One of the best ones is mypy. If you use an IDE such as PyCharm, it comes with its built-in type checker, right out of the box.

> **Caution**
>
> Remember even with type hints and a type checker, Python is still a dynamically type-checked language. That is why we call them *type hints*, i.e. they only provide hints for us and type checkers. They are not similar to what we have in compiled and statically-typed languages such as C/C++.

> **Tip**
>
> It is not required to use type hints in your code. However, we recommend that you get into the habit of using type hints where possible!

The syntax for type hints is simple. We use `: <type>` for variables and `-> <type>` for the return type of functions. All of the following are valid type hints for our `double()` function, which we can pick depending on our use case.

```
# Only integers please!
def double(x: int) -> int:
    return x * 2
```

```
# We accept integers, strings, or lists. The symbol `|` corresponds to the union of types.
def double(x: int | str | list) -> int | str | list :
    return x * 2
```

```
# Only list of integers, please!
def double(x: list[int]) -> list[int]:
    return x * 2
```

```
from typing import Any

# We accept anything and everything!
def echo(x: Any) -> Any:
    return x
```

Type hints are also supported for user-defined and custom types, e.g. classes,

```
class Book:           # `Book` is both a class and a user-defined type.
    pass

def func() -> Book:   # The type hint for the return value is `Book`.
    book = Book()
    return book
```

```
# `print()` does not return anything.
# We do not return anything, hence the return type is `None`!
def my_print(x: Any) -> None:
    print("This is awesome: ",  x)
```

```
from typing import Never

# We never return, we do not even return `None`! We fail at evaluating `1/0`!
def function_that_fails() -> Never:
    return 1/0

function_that_fails()
```

```
ZeroDivisionError: division by zero
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[26], line 7
      4 def function_that_fails() -> Never:
      5     return 1/0
----> 7 function_that_fails()
Cell In[26], line 5, in function_that_fails()
      4 def function_that_fails() -> Never:
----> 5     return 1/0
ZeroDivisionError: division by zero
```

Caution

If you do not explicitly use a type checker, Python interpreter will ignore the type hints.

```
# Only integers, please!
def double(x: int) -> int:
    return x * 2

# Python interpreter does not complain! We need to install a type checker.
double("echo!")
```

```
'echo!echo!'
```

2.2 Identities of objects

Now that we have touched upon types in Python, it is worth mentioning briefly, how objects are stored in memory in Python. As we mentioned earlier, everything in Python is an object.

> **Caution**
>
> In Python, variables are **references** to objects in memory, not the objects themselves.

> **Tip**
>
> The built-in function `id()` provides the **identity** of an object. It is called identity because it **identifies** the object during its lifetime. The identity of an object is of type `int` and is guaranteed to be unique and constant. In **CPython**, one can think of an object identity as its **memory address**. Objects with the same identities are the same objects and identical.

```
x = "Hello"

# The identity is a unique and a constant integer for every object during its lifetime.
id(x)
```

```
140352485273984
```

> **Caution**
>
> An assignments such as `y=x` simply copies the identity of `x` to `y` and not the content of `x` itself. That is why we have called `y` an **alias** of `x` in the first place, i.e. it is the **same object** as `x`!

```
x = 1.0
y = x

# `x` and `y` are the same object!
print(id(x))
print(id(y))
```

```
140355438249776
140355438249776
```

> **Caution**
>
> Note that aliasing is different from making two objects of the same type and the same value. Python **might** make new objects with different identities in this case.

```
x = 1.0
y = 1.0

# `z` is an alias of `y`.
z = y

# `x` and `y` are not the same object, although they have the same value and type.
print(id(x))
print(id(y))

print("---")

# `z` and `y` are the same object.
print(id(y))
print(id(z))
```

```
140355438252752
140355438250064
---
140355438250064
140355438250064
```

> **Caution**
>
> This is not an issue for immutable objects. However, if you make an alias of a mutable object, such as a list and manipulate the alias, you might get unwanted results. See Section 6.1.1 to know more about this.

```
x = [1, 2]
y = x

y.append(3)  # Append `3` to the end of list.
print(y)     # `y` is now [1, 2, 3].

print(x)     # `x` has also changed! Since `y` and `x` are the same object.
```

```
[1, 2, 3]
[1, 2, 3]
```

2.2.1 Equality of objects

> **Caution**
>
> In Python we have two ways of defining equality, via the `==` operator or `is` keyword. The former checks the values, whereas the latter checks the object identities.

```
x = 1.0
y = 1.0

print(id(x))
print(id(y))   # The identities are not the same.
```

```
140355438249840
140352397042256
```

```
print(x == y)  # `True` as the values are equal.
print(x is y)  # `False`, because `id(x)` differs from `id(y)`.
```

```
True
False
```

2.2.2 Interning

> **Note**
>
> As a mechanism for memory optimization, Python does **interning** for some **immutable** objects. Interning means storing only one copy of objects when possible.

> **Tip**
>
> Some objects are always interned, e.g. `True`, `False`, `None`. Other objects can be also interned depending on the case, e.g. small integers `[-5, 256]`, small strings, and small tuples.

> **Tip**
>
> For `None` we always use `is` for checking equality instead of `==`.

```
x = None
x is None  # Good!
```

```
True
```

```
x == None  # Bad!
```

```
True
```

```
x = -5
y = -5

print(x == y)
print(id(x) == id(y))   # Small integers [-5, 256] are interned.
print(x is y)
```

```
True
True
True
```

```
x = 256
y = 256

print(x == y)
print(id(x) == id(y))   # Small integers [-5, 256] are interned.
print(x is y)
```

```
True
True
True
```

```
x = 2000
y = 2000

print(x == y)
print(id(x) == id(y))  # Larger integers are not interned!
print(x is y)
```

```
True
False
False
```

> Caution
>
> Floats are never interned.

```
x = 0.0
y = 0.0

print(x == y)
print(id(x) == id(y))  # Floats are never interned.
print(x is y)
```

```
True
False
False
```

```
a = "hello"
b = "hello"

print(a == b)
print(id(a) == id(b))  # Small strings are interned.
print(a is b)
```

```
True
True
True
```

```
# Repetition of `echo` for 10000 times!
a = "echo" * 10000
b = "echo" * 10000

print(a == b)
print(id(a) == id(b))  # Larger strings are not interned.
print(a is b)
```

```
True
False
False
```

> Caution
>
> Mutable objects such as lists are never interned, even if they are small!

```
a = [1, 2]
b = [1, 2]

print(a == b)
print(id(a) == id(b))
print(a is b)
```

```
True
False
False
```

3 Numerics

Python has built-in support for integer, rational, and complex numbers. We only focus on the first two for this course.

The rational numbers are represented by floating-point numbers, aka floats. The type names for integers and floats are unsurprisingly `int` and `float`. More precisely, integers are instances (objects) of the `int` class and floats are all instances of the `float` class.

3.1 Integers

```
# an integer
2000000
```

```
2000000
```

> **Tip**
>
> One can use underscores (_) with integers and floats to improve readability. The positions and total number of underscores do not matter! The only rules are that the number cannot start or end with _ and in case of floats it cannot immediately follow or precede the decimal point.

```
# Underscores only help with readability.
2000_000
```

```
2000000
```

```
# Positions and the total number of underscores do not matter.
2_0_00_000
```

```
2000000
```

```
# A leading underscore is not okay!
_23
```

```
NameError: name '_23' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 # A leading underscore is not okay!
----> 2 _23
NameError: name '_23' is not defined
```

```
# A trailing underscore is not okay!
23_
```

```
SyntaxError: invalid decimal literal (692117513.py, line 2)
  Cell In[5], line 2
    23_
      ^
SyntaxError: invalid decimal literal
```

3.2 Floats

```
# the floating-point zero
0.0
```

```
0.0
```

> **Caution**
>
> The floating-point zero `0.0` is different from integer zero `0`, although their comparison yields `True`. The same is true about other small integers and their float counterparts. However, this is not true about larger integers.

```
0 == 0.0
```

```
True
```

```
119 == 119.0
```

```
True
```

```
# Watch out for large integers!

# The equality does not hold any more!
32459238745927123 == 32459238745927123.0
```

```
False
```

```
# One can omit the trailing zeros.
1.
```

```
1.0
```

```
# One can also omit the leading zeros.
.23
```

```
0.23
```

```
# underscores for readability
1_23.456_7_8_9
```

```
123.456789
```

```
# An underscore cannot immediately precede a decimal point.
2_.1
```

```
SyntaxError: invalid decimal literal (1979414022.py, line 2)
  Cell In[13], line 2
    2_.1
     ^
SyntaxError: invalid decimal literal
```

```
# An underscore cannot immediately follow a decimal point.
2._1
```

```
SyntaxError: invalid decimal literal (3252860201.py, line 2)
  Cell In[14], line 2
    2._1
     ^
SyntaxError: invalid decimal literal
```

> **Tip**
>
> You can use `e` or `E` for scientific notation.

```
1.23e-8
```

```
1.23e-08
```

```
1.23E-8
```

```
1.23e-08
```

3.3 Operations on numerics

Numerics support many mathematical operations as shown in the examples below.

```
x = 7
y = 3
```

```
# addition
x + y
```

```
10
```

```
# subtraction
x - y
```

```
4
```

```
# multiplication
x * y
```

21

```
# exponentiation
x ** y
```

343

```
# division, the result is always float!
x / y
```

2.3333333333333335

```
# division, but the quotient is floored and is of type `int`
x // y
```

2

```
# remainder of x / y
x % y
```

1

```
# absolute value
abs(-x)
```

7

> **Tip**
>
> Like C/C++, Python supports augmented assignment operators, e.g. `+=` and `/=`. However, unlike C/C++, Python does not have increment (`++`) or decrement (`--`) operators.

```
x = 5
x += 1

print(x)
```

6

```
x = 5

# This does not work! Python does not have `++`.
x++
```

```
SyntaxError: invalid syntax (4082388416.py, line 4)
  Cell In[27], line 4
    x++
      ^
SyntaxError: invalid syntax
```

3.4 Conversion of numeric types

Both `int` and `float` can convert from a number of different input types.

```
int(2.3)
```

```
2
```

```
int(2.8)
```

```
2
```

```
int(-2.3)
```

```
-2
```

```
float(23)
```

```
23.0
```

```
float("1.2")
```

```
1.2
```

Caution

In case of addition, multiplication, subtraction, and exponentiation, the result is of type `int` if all operands are also of type `int`. The only exception is division `/` which always returns `float`. This is in contrast to languages such as C/C++, where the result of dividing two integers is also an integer.

Tip

If you want the result of the division to be integer use `//` instead.

```
# The result is of type `int`, since all operands are integers.
type(2 + 3*4 + 2**3 - 10)
```

```
int
```

```
# The result is of type `float`, unlike languages such as C/C++.
type (6 / 3)
```

```
float
```

```
# The result is of type `int`, since the quotient is floored with `//`.
type (6 // 3)
```

```
int
```

When there is a mix of integers and floats, the results are always implicitly **promoted** to `float`.

```
2 + 3.0
```

```
5.0
```

> **Caution**
>
> Type **demotion** (float to int) is not generally safe, as we lose data! The situation is slightly better for type **promotion** (int to float), but there are still cases that we need to watch out for, e.g. very large integers.

```
x = 12324234
y = float(x)

# We are okay! The values of `x` and `y` are equal.
print(y)
print(x == y)
```

```
12324234.0
True
```

```
x = 3.14
y = int(x)

# We are not okay! The values are not equal, as we have lost some data!
print(y)
print(x == y)
```

```
3
False
```

```
# a large random integer
x = 19824719028734201
y = float(x)

# Since `x` is a large integer, the values of `x` and `y` are no longer equal!
print(y)
print(x == y)
```

```
1.98247190287342e+16
False
```

3.5 Booleans

Booleans are also numerics and a subclass of `int`. However, there are only two instances of type `bool` in Python. They are `True` and `False`. Booleans appear in all logical operations.

```
# Booleans are instances of integers.
isinstance(True, int)
```

```
True
```

```
isinstance(False, int)
```

```
True
```

> **Caution**
>
> In Python, we use `and`, `or`, and `not` for logical operations. There are also `&` and `|` operators, however, they are for bitwise operations.

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
not True
```

```
False
```

```
x = None
x is None
```

```
True
```

```
3 > 2
```

```
True
```

```
4 <= 5 <= 7  # We can chain comparisons!
```

```
True
```

```
2 == 4
```

```
False
```

```
"A" in "Apples"
```

```
True
```

4 Collections

A Python collection is an object that contains zero or more elements. Collections can be either **unordered** or **ordered**.

> **Tip**
>
> Ordered collections are specifically referred to as **sequences**. Examples of sequences are strings (Chapter 5), lists (Section 6.1), and tuples (Section 6.2). Examples of unordered collections are sets (Section 7.1) and dictionaries (Section 7.2).

> **Caution**
>
> Sets (Section 7.1) and dictionaries (Section 7.2) are **not ordered** and therefore not sequences.

> **Tip**
>
> You can use the built-in function `issubclass()` alongside `collections.abc.Collection` or `collections.abc.Sequence` to check whether an object is a collection or a sequence.

```
from collections.abc import Collection
from collections.abc import Sequence

issubclass(Sequence, Collection)      # All sequences are collections.
```

```
True
```

```
issubclass(Collection, Sequence)      # But not the other way around.
```

```
False
```

```
print(issubclass(list, Collection))
print(issubclass(list, Sequence))
```

```
True
True
```

```
print(issubclass(str, Collection))
print(issubclass(str, Sequence))
```

```
True
True
```

```
print(issubclass(tuple, Collection))
print(issubclass(tuple, Sequence))
```

```
True
True
```

```
print(issubclass(set, Collection))
print(issubclass(set, Sequence))
```

```
True
False
```

```
print(issubclass(dict, Collection))
print(issubclass(dict, Sequence))
```

```
True
False
```

> **Caution**
>
> One can also categorize collections based on their **mutability** or **immutability**. Among the collections we have discussed,
> - mutable collections:
>   - lists
>   - sets
>   - dictionaries
> - immutable collections:
>   - strings
>   - tuples

4.1 Collections are sized

Collections have a length which determines the number of their elements. Collections can be empty, however, their maximum allowed length is determined by the available memory.

> **Tip**
>
> Python built-in function `len()` is used to get the number of elements in a collection.

> **Note**
>
> Collections inherit from Sized and implement the `__len__` magic method.

```
sequence_str = "Hello!"

print(sequence_str)
print(type(sequence_str))
print(len(sequence_str))
```

```
Hello!
<class 'str'>
6
```

```
sequence_list = [1, 2.0, "hello", print, [3, 4]]

print(sequence_list)
print(type(sequence_list))
print(len(sequence_list))
```

```
[1, 2.0, 'hello', <built-in function print>, [3, 4]]
<class 'list'>
5
```

```
sequence_tuple = (1, 2, "hello")

print(sequence_tuple)
print(type(sequence_tuple))
print(len(sequence_tuple))
```

```
(1, 2, 'hello')
<class 'tuple'>
3
```

```
collection_set = {1, 1, 1, 2, 3}  # Repetitions are removed in a set.

print(collection_set)
print(type(collection_set))
print(len(collection_set))
```

```
{1, 2, 3}
<class 'set'>
3
```

```
collection_dict = {"book": 10, "apples": 20}

print(collection_dict)
print(type(collection_dict))
print(len(collection_dict))
```

```
{'book': 10, 'apples': 20}
<class 'dict'>
2
```

4.1.1 Truth value of collections

> **Tip**
>
> Empty collections evaluate to `False` in a boolean context.

```
s = ""

if s:
    print(s)
else:
    print("I am empty!")
```

```
I am empty!
```

```
lst = [1, 2, 3]

if lst:
    print("I am not empty!")
else:
    print("I am empty!")
```

```
I am not empty!
```

```
d = {}   # `{}` is an empty dictionary.
not d
```

```
True
```

4.2 Collections are containers

> **Tip**
>
> One can check if an object belongs to a collection. This is done by using the `in` keyword.

> **Note**
>
> Collections inherit from Container and implement the `__contains__` magic method.

```
"h" in "hello"
```

```
True
```

```
2 in {5, 3, 1}
```

```
False
```

4.3 Collections are iterable

Iteration over a collection means we access each element of the collection one at a time, typically using a loop. For sequences the **order is preserved** in iteration. For unordered collections, however, the order is **not guaranteed**.

> **Note**
>
> Collections inherit from Iterable and implement the `__iter__` magic method.

```
sequence_str = "Hello"
for letter in sequence_str:     # The order is preserved, since strings are sequences.
    print(letter)
```

```
H
e
```

```
l
l
o
```

```
collection_set = {3, 5, 1}
for element in collection_set:  # The order is not preserved, since sets are unordered!
    print(element)
```

```
1
3
5
```

4.4 Indexing and slicing of sequences

Owing to being ordered, elements in a sequence can be accessed and referred to by indices enclosed in square brackets `[]`. This is similar to what we do with vectors, arrays, or matrices in Mathematics.

Python, like C/C++ and Java, uses zero-based numbering for sequences. This means the first (**actually the zeroth**) element of a sequence has an index of `0`. This is in contrast to e.g. Matlab which uses one-based numbering.

```
s = "hello!"
print(s[0])
```

```
h
```

```
s = {1, 2, 3}
print(s[0])      # What does even the zeroth element mean for a set?
```

```
TypeError: 'set' object is not subscriptable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[21], line 2
      1 s = 1, 2, 3
----> 2 print(s[0])      # What does even the zeroth element mean for a set?
TypeError: 'set' object is not subscriptable
```

> **Note**
>
> For a sequence S, a valid index i satisfies the following condition
>
> $$i \in \mathbb{Z} \quad \text{and} \quad -N \leq i \leq N-1;$$
>
> where $N = \text{len}(S)$ is the length of the sequence and $\mathbb{Z}$ is the set of all integers.
> An index of -1 corresponds to the last item in the sequence, -2 to the penultimate item, etc.

```
s = ["a", "b", "c"]
print(s[0], s[1], s[2])
print(s[-1], s[-2], s[-3])
```

```
a b c
c b a
```

> **Caution**
>
> Indices must be integers. Floats, even if they are equal to an integer, are not accepted as indices.

```
s[2]    # Okay!
s[2.0]  # Not okay! `2.0` is not of type `int`, although `2 == 2.0` yields `True`.
```

```
TypeError: list indices must be integers or slices, not float
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 2
      1 s[2]    # Okay!
----> 2 s[2.0]  # Not okay! `2.0` is not of type `int`, although `2 == 2.0` yields `True`.
TypeError: list indices must be integers or slices, not float
```

4.4.1 Slicing

> **Tip**
>
> One can retrieve a range of elements from the sequence using the `[<first_index>:<last_index>]` notation, e.g. `s[3:7]`.

> **Caution**
>
> Note that `<first_index>` is inclusive but `<last_index>` is exclusive.

```
s = "abcd"

s[1:3]     # The index `3` is exclusive, so "d" is not printed.
```

```
'bc'
```

> **Tip**
>
> You can omit `<first_index>`, `<last_index>`, or both.

```
s = "Apples and Oranges!"

print(s[:6])    # First index is implicitly set to `0`.
print(s[7:11])  # Both indices are present.
print(s[11:])   # Last index is omitted, which means retrieve all the rest!

print(s[:])     # This is essentially the same as `print(s)`.
```

```
Apples
and
Oranges!
Apples and Oranges!
```

> **Caution**
>
> Note that `[<first_index>:]` is not the same as `[<first_index>:-1]`. The last index is exclusive. See the example below.

```
s = "Apples"

print(s[0:])
print(s[0:-1])   # The last index of `-1` is exclusive, therefore `s` is not printed.
```

```
Apples
Apple
```

> **Tip**
>
> You can increment indices by values other than `1`, using the `[<first_index>:<last_index>:<step>]` notation, e.g. `s[2:11:3]`.

```
s = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(s[0::2])  # odd numbers
print(s[1::2])  # even numbers
```

```
[1, 3, 5, 7, 9]
[2, 4, 6, 8]
```

```
s = "abcdefghijklmnopqrstuvwxyz"

print(s[::-1])  # An interesting one-liner to reverse a string!
```

```
zyxwvutsrqponmlkjihgfedcba
```

5 Strings

They are immutable and (ordered) sequences of Unicode characters. A string must be enclosed by either single (') or double quotations ("), but not a mix of the two. There is no convention on whether to use single or double quotations. However, consistency helps with readability.

> **Note**
>
> We recommend that you use double quotations where possible.

> **Note**
>
> In contrast to C/C++, Python does not differentiate between single characters which are of type `char` and enclosed in single quotations; and a sequence of characters, e.g. `char *`, `char []`, or `std::string`, enclosed in double quotations.

5.1 Examples

```
'I am enclosed in single quotes.'
```

```
'I am enclosed in single quotes.'
```

```
"I am enclosed in double quotes."  # Note the output is printed in single quotes!
```

```
'I am enclosed in double quotes.'
```

5.1.1 Multi-line strings

> **Tip**
>
> For multi-line strings, you have two options
> - Use the newline escape sequence, i.e `\n`.
> - Enclose the string in triple single quotes `'''` or triple double quotes `"""`, and split the string over multiple lines.

> **Note**
>
> We refer to both *triple single quotes* and *triple double quotes* as **triple quotes**.

```
"I am a multi-line string!\nThis is the second line!"  # Use `print()` to see the effect!
```

```
'I am a multi-line string!\nThis is the second line!'
```

```
print("I am a multi-line string!\nThis is the second line!")  # There we go!
```

```
I am a multi-line string!
This is the second line!
```

```
text = """I am a multi-line string!
This is the second line."""

print(text)
```

```
I am a multi-line string!
This is the second line.
```

```
text = '''I am also a multi-line string but enclosed in triple single quotes!
This is the second line!'''

print(text)
```

```
I am also a multi-line string but enclosed in triple single quotes!
This is the second line!
```

> **Note**
>
> Triple quotes signify a multi-line string only if the string spans over more than one line, or it includes the newline sequence \n. Otherwise, it is interpreted as a single-line string.

```
text = """Although using triple quotes, I am still a single-line string."""

print(text)
```

```
Although using triple quotes, I am still a single-line string.
```

```
text = """
----------
Unlike me!
----------
"""

print(text)
```

```
----------
Unlike me!
----------
```

```
print("""or \n me!""")
```

```
or
 me!
```

5.1.2 Docstrings

> **Tip**
>
> In python, # is used for single-line comments. This means in case of multi-line comments, each line must start with #. As an alternative one can also use multi-line strings with triple quotes. Such multi-line strings (in place of comments) are mainly used for documentation purposes and are called **docstrings**.

```
# I am the first line of the comment,
# and I'm the second line.

def area_of_circle(r):
    """Calculate the area of a circle given its radius.

    We use the value of Pi to four decimal places, i.e. `pi = 3.1416`.

    Note that I am a multi-line string with triple quotes which is
    used for documentation. I am known as a `docstring`.
    """
    return 3.1416 * r**2

area_of_circle(2.0)
```

```
12.5664
```

5.1.3 How to mix and match!?

There are occasions when the string itself includes double or single quotes. Examples of such cases are

- *I'm reading this now!*
- *"This seems to be a chicken-and-egg problem," said the researcher!*

To construct such strings in Python, you can

- Enclose the string in the other type of quotes.
- Escape the internal quotes, i.e \' or \", similar to what is done in languages such as C/C++.
- Use triple quotes.

Examples of mixing single and double quotes

```
"I'm reading this now!"
```

```
"I'm reading this now!"
```

```
'"This seems to be a chicken-and-egg problem," said the researcher!'
```

```
'"This seems to be a chicken-and-egg problem," said the researcher!'
```

```
'I\'m reading this now!'  # escape sequence
```

```
"I'm reading this now!"
```

```
"\"This seems to be a chicken-and-egg problem,\" said the researcher!"  # escape sequence
```

```
'"This seems to be a chicken-and-egg problem," said the researcher!'
```

5.2 Indexing and iteration over strings

As we explained in Chapter 4, strings are sequences. As a result, they can be both indexed and iterated over.

> **Warning**
>
> Strings are immutable.

```
text = "Apples and Oranges!"
text[-1]
```

```
'!'
```

```
# This does not work! Strings are immutable.
text[2] = "a"
```

```
TypeError: 'str' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 2
      1 # This does not work! Strings are immutable.
----> 2 text[2] = "a"
TypeError: 'str' object does not support item assignment
```

5.3 Different kinds of strings

Depending on our use case, Python offers a number of different string kinds,

- Unicode (regular) strings
- Raw strings
- Byte strings
- Formatted strings, aka *f-strings*

5.3.1 Unicode strings

Regular strings in Python are essentially Unicode strings, right out of the box. All the examples we have given earlier correspond to Unicode strings.

5.3.2 Raw strings

As you know, backslashes (\) can have two meanings in the context of strings

- A literal character, i.e. a single \
- An escape character to give specific meanings to the proceeding character, e.g.,
  - \n newline
  - \t tab
  - \b backspace

> **Caution**
>
> In Unicode strings backslashes are by default treated as a escape character.

```
print("This is\ninteresting\tand\tfun!")
```

```
This is
interesting and fun!
```

As a result, if we want to have a literal backslash in a string we should escape it with another backslash. This means for every literal backslash we need two backslashes \\ in the string. This can quickly become confusing and error-prone as the number of backslashes increases. This is especially important when strings include instructions of a programming language or file paths. For example on Windows, different parts of a path are separated by \.

Raw strings are a solution to this confusion. They are prefixed with r or R and automatically escape the backslash, i.e. backslashes are treated as literal characters and not as escape characters anymore.

```
print("a\tb\tc")    # \t is the tab character
print("a\\tb\\tc")  # \\ indicates a literal backslash, this is confusing!
```

```
a   b   c
a\tb\tc
```

```
print(r"a\tb\tc")  # \ indicates a literal backslash, better!
```

```
a\tb\tc
```

```
print("Navigate to C:\new\table\n1.dat")   # This is not even what we want!
```

```
Navigate to C:
ew  able
1.dat
```

```
print("Navigate to C:\\new\\table\\n1.dat")  # This is what we want but it is confusing.
```

```
Navigate to C:\new\table\n1.dat
```

```
print(r"Navigate to C:\new\table\n1.dat")    # This is the best!
```

```
Navigate to C:\new\table\n1.dat
```

5.3.3 Byte strings

Byte strings represent sequences of raw bytes rather than Unicode characters. They are prefixed with b or B. They can be decoded given an encoding such as utf-8 and the .decode() method called on the string instance. By the same token, Unicode strings can be encoded into byte strings using the .encode() method.

> **Tip**
>
> Byte strings are useful when reading binary data from e.g. a file.

```
data = b"\x48\x65\x6c\x6c\x6f"  # ASCII bytes for "Hello"
print(data)                     # Note the `b` prefix.
```

```
b'Hello'
```

```
unicode_string = data.decode("utf-8")

print(unicode_string)
```

```
Hello
```

```
byte_string = unicode_string.encode("utf-8")

print(byte_string)
```

```
b'Hello'
```

```
data == byte_string  # We test to see if they are in fact equal.
```

```
True
```

5.3.4 Formatted strings

Formatted strings or f-strings allow for literal string interpolation i.e. they embed expressions inside themselves. The expressions will be evaluated and replaced accordingly. They are prefixed with `f` or `F` and expressions are enclosed in braces `{}`.

```
course = "TNM119"
year = 2025
text = f"{course} in {year}"

print(text)
```

```
TNM119 in 2025
```

```
x = 2
y = 3

print(f"The sum of x and y is {x+y}")
```

```
The sum of x and y is 5
```

6 Lists and tuples

6.1 Lists

Lists are (ordered) sequences of objects. Items in a list can be of different types and do not need to be unique. They are enclosed in square brackets `[]` where comma `,` separates the different elements. They can be empty or be as large as needed (subject to the available memory).

```
lst1 = []          # making a list with its elements, empty in this case!
lst2 = list()      # making a list by calling `list()`

print(lst1, lst2)
```

```
[] []
```

```
lst = [1, 1, 2, 3]  # Non-unique items are allowed.

print(lst)
```

```
[1, 1, 2, 3]
```

```
lst = [1.0, 1, "1", print]  # Different types of objects are allowed.

print(lst)
```

```
[1.0, 1, '1', <built-in function print>]
```

6.1.1 Lists are mutable

```
lst = [1, 2, 3]
print(lst)

lst[-1] = 4        # Modify the last element.
print(lst)
```

```
[1, 2, 3]
[1, 2, 4]
```

> **Caution**
>
> Since lists are mutable when you make an alias of them, and manipulate the alias, you might get unwanted results.

```
lst_original = [1, 2, 3]
lst_alias = lst_original   # We have supposedly made a copy of the original list!
lst_alias[-1] = 5

print(lst_alias)
print(lst_original)        # Oops! The original list has also changed!
```

```
[1, 2, 5]
[1, 2, 5]
```

```
# Oh! I see why.
# Both lists have the same `identity`!

print(id(lst_original))
print(id(lst_alias))
```

```
140487797517632
140487797517632
```

6.1.1.1 Shallow copying

> **Tip**
>
> To avoid unwanted mutation of the original list, you have two options
> - Use slicing `[:]`.
> - Use the `copy()` function.

Example on how to use `[:]`

```
lst_original = [1, 2, 3]
lst_copy = lst_original[:]  # Use slicing to make a (shallow) copy.
lst_copy[-1] = 5

print(lst_copy)
print(lst_original)         # Great! The original list has not changed!
```

```
[1, 2, 5]
[1, 2, 3]
```

```
# The lists have different identities!
print(id(lst_original))
print(id(lst_copy))
```

```
140487797514880
140487797510080
```

The same example with `copy()`

```
import copy

lst_original = [1, 2, 3]
lst_copy = copy.copy(lst_original[:])   # Use the `copy()` function, same as `[:]`.
lst_copy[-1] = 5

print(lst_copy)
print(lst_original)                     # Great! The original list has not changed!
```

```
[1, 2, 5]
[1, 2, 3]
```

```
# The lists have different identities!

print(id(lst_original))
print(id(lst_copy))
```

```
140487797622208
140487797509632
```

6.1.1.2 Deep copying

> **Caution**
>
> Both `[:]` and `copy()` perform a so-called **shallow copying** of the elements. This means although a new list (with a new identity) is made, the elements are still copied by reference. This is okay if elements are not mutable, e.g. they are integers. However, if the elements are mutable themselves, this approach does not work and one must use the `deepcopy()` function.

```
import copy

lst_original = [1, 2, [3, 4]]           # My last element is a list (mutable).
lst_copy = copy.copy(lst_original[:])   # Use the `copy()` function, same as `[:]`.

lst_copy[0] = 8              # Change the first element.
lst_copy[-1][-1] = 15        # Change the last element.

print(lst_copy)
print(lst_original)          # The mutable element has changed in both lists! Not good!
```

```
[8, 2, [3, 15]]
[1, 2, [3, 15]]
```

```
# But why? I thought I have copied the content.
# The lists even have different identities!

print(id(lst_original))
print(id(lst_copy))
```

```
140487797514880
140487797622144
```

```
# Oh! I see why.
# The last element of both lists have the same `identity`!

print(id(lst_original[-1]))
print(id(lst_copy[-1]))
```

```
140487797516288
140487797516288
```

Example on how to use `copy.deepcopy()`

```
import copy

lst_original = [1, 2, [3, 4]]               # My last element is a list (mutable).
lst_copy = copy.deepcopy(lst_original[:])   # Use `deepcopy()`.
lst_copy[0] = 8
lst_copy[-1][-1] = 15

print(lst_copy)
print(lst_original)          # The original list has not changed. Great!
print("---")
print(id(lst_original))
print(id(lst_copy))          # The lists have different identities!
print("---")
print(id(lst_original[-1]))
print(id(lst_copy[-1]))      # The last elements of lists have different identities. Great!
```

```
[8, 2, [3, 15]]
[1, 2, [3, 4]]
---
140487797623680
140487797507328
---
140487797507520
140487797507072
```

6.1.2 Operations on lists

> **Tip**
>
> To add a single item to the end of the list, use `.append()`.

```
lst = [1, 2, 3]
lst.append(4)    # Append 4 to the end of the list.

print(lst)
```

```
[1, 2, 3, 4]
```

> **Caution**
>
> In case of multiple items, `.append()` is not the solution. Instead use `.extend()`.

```
lst = [1, 2, 3]
lst.append([4, 5])    # Is this what we really want?

print(lst)
```

```
[1, 2, 3, [4, 5]]
```

```
lst = [1, 2, 3]
lst.extend([4, 5])    # Add 4, and 5 to the end of the list.

print(lst)
```

```
[1, 2, 3, 4, 5]
```

> **Tip**
>
> Use + for list concatenation and * for list multiplication.

```
[1, 2, 3] + [4, 5]
```

```
[1, 2, 3, 4, 5]
```

```
[1, 2, 3] * 2
```

```
[1, 2, 3, 1, 2, 3]
```

> **Caution**
>
> When used on lists, + and * are not arithmetic operators.

6.1.3 List unpacking

We can unpack a list using an asterisk * as a prefix operator.

```
lst = [1, 2, 3]

print(lst)      # Print the list.
print(*lst)     # Print the individual elements.
```

```
[1, 2, 3]
1 2 3
```

```
lst = [1, 2, 3, 4]
lst2 = [10, 11, *lst, 15]

print(lst2)
```

```
[10, 11, 1, 2, 3, 4, 15]
```

```
def sum(x, y, z):
    return x + y + z

lst = [2, 3, 6]

print(sum(lst[0], lst[1], lst[2]))  # It works, but there is a better way!
print(sum(*lst))                    # Use unpacking instead!
```

```
11
11
```

```
print(sum(lst))                     # Bug! The function needs three arguments.
```

```
TypeError: sum() missing 2 required positional arguments: 'y' and 'z'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 print(sum(lst))                     # Bug! The function needs three arguments.
TypeError: sum() missing 2 required positional arguments: 'y' and 'z'
```

6.1.4 Iteration over lists

One can use `for` loops to iterate directly over the list elements. As such, Python `for` loops resemble what are referred to as `for-each` loops in other programming languages.

```
lst = ["a", "b"]

for element in lst:  # No need for indices, we iterate over actual elements.
    print(element)
```

```
a
b
```

> **Tip**
>
> If you also need the indices, use the built-in `enumerate()` function.

```
lst = ["a", "b"]

for idx, element in enumerate(lst):
    print(idx, element)
```

```
0 a
1 b
```

> **Tip**
>
> If you want to iterate over a specific range of numbers, you can use Python built-in `range()`.

```
# All numbers starting from `0` (inclusive) to `10` (exclusive) in steps of `1`
for i in range(10):
    print(i, end=" ")
```

```
0 1 2 3 4 5 6 7 8 9
```

```
# All numbers starting from `3` (inclusive) to `10` (exclusive) in steps of `2`
for i in range(3, 10, 2):
    print(i, end=" ")
```

```
3 5 7 9
```

```
# All numbers starting from `10` (inclusive) to `3` (exclusive) in steps of `-2`
for i in range(10, 3, -2):
    print(i, end=" ")
```

```
10 8 6 4
```

> **Tip**
>
> To iterate over two (or more) lists, use the built-in `zip()` function.

```
lst1 = ["a", "b", "c"]
lst2 = [10, 20, 30]

for e1, e2 in zip(lst1, lst2):
    print(e1, e2)
```

```
a 10
b 20
c 30
```

> **Caution**
>
> In case of lists with different lengths, the output of `zip()` has a length equal to the minimum length of the input lists.

```
lst1 = ["a", "b"]     # length of 2
lst2 = [10, 12, 4]    # length of 3

for e1, e2 in zip(lst1, lst2):   # Note the lists are not of the same length!
    print(e1, e2)                # We only get two outputs, is `lst1` missing an element?
```

```
a 10
b 12
```

> **Caution**
>
> To ensure that you get an error when lists are not of the same length, set `strict=True` in `zip()`.

```
lst1 = ["a", "b"]     # length of 2
lst2 = [10, 12, 4]    # length of 3

for e1, e2 in zip(lst1, lst2, strict=True):  # Good! We know if the data is corrupted!
    print(e1, e2)
```

```
a 10
b 12
```

```
ValueError: zip() argument 2 is longer than argument 1
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[31], line 4
      1 lst1 = ["a", "b"]     # length of 2
      2 lst2 = [10, 12, 4]    # length of 3
----> 4 for e1, e2 in zip(lst1, lst2, strict=True):  # Good! We know if the data is corrupted!
      5     print(e1, e2)
ValueError: zip() argument 2 is longer than argument 1
```

> **Tip**
>
> You can use the `zip()` function to iterate over any of number of lists.

```
lst1 = ["a", "b", "c"]
lst2 = [10, 12, 4]
lst3 = [-1, -2, -3]
lst4 = ["/", "?", "@"]

for e1, e2, e3, e4 in zip(lst1, lst2, lst3, lst4, strict=True):
    print(e1, e2, e3, e4)
```

```
a 10 -1 /
b 12 -2 ?
c 4 -3 @
```

6.1.5 List comprehension

You can iterate over a list, do some operations on individual elements, and assign the results to a new list. This can be best achieved by list comprehension instead of normal `for` loops.

```
# This implementation is not optimal.

lst = [2, 3, 5]

result = []
for element in lst:
    result.append(element * 2)

result
```

```
[4, 6, 10]
```

```
lst = [2, 3, 5]

result = [2 * element for element in lst]  # list comprehension!

result
```

```
[4, 6, 10]
```

> **Tip**
>
> Using list comprehension over `for` loops is preferred. However, it is better to avoid making overly complex list comprehensions! Simplicity and readability matters.

> **Tip**
>
> You can include `if` in list comprehensions.

```
# Example of a useful list comprehension

lst = [2, 1, 7, 4, 8, 6, 3, 5, 9, 10]

even = [element for element in lst if element % 2 == 0]   # Extract even numbers.

even
```

```
[2, 4, 8, 6, 10]
```

6.2 Tuples

Tuples are like lists but they are immutable. Moreover, they are enclosed in parentheses `()` instead of `[]`.

```
t1 = ()        # making a tuple with its elements, empty in this case!
t2 = tuple()   # making a tuple by tuple()

print(t1, t2)
```

```
() ()
```

```
t = (1, 2)

print(t)
print(t[-1])
```

```
(1, 2)
2
```

> **Tip**
>
> You can omit parentheses, when making a tuple from its elements, this is referred to as *tuple packing*.

```
t = 1, 2, 3    # This is a tuple.

print(t)
```

```
(1, 2, 3)
```

> **Caution**
>
> Tuples are immutable. You cannot modify their elements and add or remove elements from them. If you want to change anything you must make a new tuple.

```
t = 1, 2, 3

t[-1] = 10    # This is not allowed!
```

```
TypeError: 'tuple' object does not support item assignment
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[39], line 3
      1 t = 1, 2, 3
----> 3 t[-1] = 10    # This is not allowed!
TypeError: 'tuple' object does not support item assignment
```

```
t = 1, 2, 3
t = t[0], t[1], 10  # Define a new tuple, instead.

print(t)
```

```
(1, 2, 10)
```

Similar to tuple packing, tuple unpacking can be performed to retrieve and assign individual elements of the tuples.

```
a, b = 1, 2  # The right-hand side is a tuple.

print(a)
print(b)
```

```
1
2
```

```
a, b, *c = 1, 2, 3, 4, 5

print(a)
print(b)
print(c)   # `c` is a list, interesting!
```

```
1
2
[3, 4, 5]
```

> **Tip**
>
> You can use a single underscore (_) as a placeholder for a value that you do not need.

```
a, _, *c, d, _ = 1, 2, 3, 4, 5, 6, 7  # We skip 2 and 7 with _

print(a)
print(c)
print(d)
```

```
1
[3, 4, 5]
6
```

> **Note**
>
> We can use a one-liner to swap the values of two variables in Python. See the example below. Can you explain how it works?

```
x = 10
y = 20
print(x, y)

x, y = y, x  # Swap the values.

print(x, y)
```

```
10 20
20 10
```

> **Note**
>
> The `zip()` function we described before returns an iterator of tuples.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [7, 8, 9]

for element in zip(a, b, c):
    print(element, type(element))
```

```
(1, 4, 7) <class 'tuple'>
(2, 5, 8) <class 'tuple'>
(3, 6, 9) <class 'tuple'>
```

7 Sets and dictionaries

7.1 Sets

Sets are mutable (unordered) collections. However, all elements in a set must be unique. This is in agreement with the Mathematical definition of a set. Elements in a set are enclosed in curly braces {}.

```
s1 = {1, 2, 3}  # making a set with its elements.
s2 = set()      # making a set by calling `set()`

print(s1)
print(s2)
```

```
{1, 2, 3}
set()
```

```
s = {1, 1, 1}   # Repetitions are discarded.

print(s)
```

```
{1}
```

> **Caution**
>
> Unlike lists and tuples, empty sets cannot be made by assigning to {}. This is because Python interprets empty {} as a dictionary (Section 7.2) and not a set.

```
l = []  # list
t = ()  # tuple
d = {}  # dictionary

print(type(l))
print(type(t))
print(type(d))
```

```
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

> **Caution**
>
> Elements of a set must be hashable. As a result, a set cannot include e.g. a list, since lists are not hashable.

```
s = {[1]}
```

```
TypeError: unhashable type: 'list'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 s = [1]
TypeError: unhashable type: 'list'
```

> **Tip**
>
> Since Python sets resemble their Mathematical counterparts, they support set operations, i.e. subtraction (`-`), intersection (`&`) and union (`|`).

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
s3 = {5, 6, 7}

print(s1 & s2)  # intersection
print(s1 | s2)  # union
print(s2 - s1)  # subtraction
print(s1 - s2)  # subtraction
```

```
{2, 3}
{1, 2, 3, 4}
{4}
{1}
```

> **Tip**
>
> You can add a new element to a set using the union notation or using the `.add()` method.

```
s = {1, 2, 3}
s.add(4)
print(s)

s = s | {5}
print(s)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
```

> **Caution**
>
> Since sets are not sequences, they cannot be indexed. However, they are still iterable.

```
s = {"a", "b", "c"}

for element in s:    # Iteration is allowed
    print(element)
```

```
a
```

```
b
c
```

```
s[-1]                # But not indexing.
```

```
TypeError: 'set' object is not subscriptable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 s[-1]                # But not indexing.
TypeError: 'set' object is not subscriptable
```

> **Tip**
>
> Like lists, sets also support comprehensions.

```
s = {2, 1, 7, 4, 8, 6, 3, 5, 9, 10}

even = {element for element in s if element % 2 == 0}  # Set of even numbers.

print(even)
```

```
{2, 4, 6, 8, 10}
```

7.2 Dictionaries

Dictionaries are unordered collections of key-value pairs, aka *items*. In other words, a dictionary maps a collection of keys to a collection of values. As a result, the number of keys and values in a dictionary are the same. Dictionaries are enclosed in curly braces `{}` like sets. For each item, the key and its corresponding value are separated by a single colon `:` and comma `,` separates the items.

```
d1 = {"apple": "red", "banana": "yellow"}   # making a dictionary with its elements
d2 = dict(apple="red", banana="yellow")     # making a dictionary by calling `dict()`

print(d1)
print(d2)
```

```
{'apple': 'red', 'banana': 'yellow'}
{'apple': 'red', 'banana': 'yellow'}
```

> **Caution**
>
> Using `dict(<key>=<value>)` for making a dictionary only works if the keys are strings and valid identifier names in Python.

```
d1 = {"apple !": "a"}      # Okay!
print(d1)
```

```
{'apple !': 'a'}
```

```
d2 = dict(apple !="a")     # Not okay! `apple !` is not a valid Python identifier.
```

```
NameError: name 'apple' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 d2 = dict(apple !="a")     # Not okay! `apple !` is not a valid Python identifier.
NameError: name 'apple' is not defined
```

```
d = {}           # This is an empty dictionary not an empty set!

print(type(d))
```

```
<class 'dict'>
```

> **Note**
>
> The corresponding value of a key can be retrieved by `[<key>]`.

```
d = {"apple": "red", "banana": "yellow"}

d["apple"]
```

```
'red'
```

> **Note**
>
> Keys of a dictionary are like sets (Section 7.1), i.e they need to be unique and hashable. As a result, lists cannot be keys of a dictionary. Values, however, can be mutable, non-unique and of different types.

```
d = {"b": "book", "a": []}   # Okay! Keys are integers.

d["a"].append("apple")       # Values can be mutable.

print(d)
```

```
{'b': 'book', 'a': ['apple']}
```

```
d = {[]: "a"}   # Not okay! Lists are not hashable and cannot be keys.
```

```
TypeError: unhashable type: 'list'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 d = []: "a"   # Not okay! Lists are not hashable and cannot be keys.
TypeError: unhashable type: 'list'
```

> **Caution**
>
> When using `dict()`, repetition of keys are not allowed. However, if you use `{}`, you can have repeated keys, but only the last one will be kept.

```
# Repeated keys are not allowed when using `dict()`.

d = dict(a=1, a=2)

print(d["a"])
```

```
SyntaxError: keyword argument repeated: a (4085247928.py, line 3)
  Cell In[17], line 3
    d = dict(a=1, a=2)
                  ^
SyntaxError: keyword argument repeated: a
```

```
# Since keys must be unique, only the last item (with the same key) will be kept.

d = {"a": 1, "a": 2}

print(d["a"])
```

```
2
```

7.2.1 Iteration over dictionaries

There are a number of different ways to iterate over a dictionary depending on our use case.

```
sample = {"a": "apple", "b": "book", "c": "car"}
```

```
# Iterate over keys.

for key in sample.keys():
    print(key, sample[key])
```

```
a apple
b book
c car
```

```
# Iterate over values.

for value in sample.values():
    print(value)
```

```
apple
book
car
```

```
# Iterate over items.

for key, value in sample.items():
    print(key, value)
```

```
a apple
b book
c car
```

> **Tip**
>
> When iterating over keys, one can omit `.keys()`.

```
for key in sample:            # Same as `for key in sample.keys()`
    print(key, sample[key])
```

```
a apple
b book
c car
```

> **Tip**
>
> Like lists and sets, dictionaries support comprehensions.

```
sample = {"a": 1, "b": 2, "c": 7, "d": 25, "e": 4}
even = {key: value for key, value in sample.items() if value % 2 == 0}

print(even)
```

```
{'b': 2, 'e': 4}
```

> **Caution**
>
> Dictionaries support the union operation `|` but not intersection or subtraction. However, the order matters, i.e. the result of `d1 | d2` can be different from that of `d2 | d1`.

```
d1 = {"a": "apple", "b": "book", "c": "car"}
d2 = {"b": "bicycle", "d": "drawer"}

print(d1 | d2)      # The value of "b" becomes "bicycle".
print(d2 | d1)      # The value of "b" becomes "book".
```

```
{'a': 'apple', 'b': 'bicycle', 'c': 'car', 'd': 'drawer'}
{'b': 'book', 'd': 'drawer', 'a': 'apple', 'c': 'car'}
```

8 Functions

8.1 Function signature and body

In Python functions are also objects. They allow you to structure the code in smaller units, where each unit ideally does one specific job. Functions are defined by `def` keyword, followed by the function signature, a single colon `:`, and finally the function body.

```
def <function_signature>:
    <function_body>
```

or

```
def <function_signature>: <function_body>
```

e.g.

```
def multiply(x, y):
    return x * y
```

```
def multiply(x, y): return x * y
```

or with (optional) type hints

```
def multiply(x: float, y: float) -> float:
    return x * y
```

> **Note**
>
> The function signature consists of the *function name*, *function parameters* inside parentheses, and optionally the type hints for parameters and the return value. In the above example, both `multiply(x, y)` and `multiply(x: float, y: float) -> float` are function signatures.

> **Tip**
>
> We follow the **snake_case** convention for function names.

> **Caution**
>
> The function body can be on the same line as the function signature. This is not recommended unless the function body is trivial, such as `return x * y` shown above.

> **Note**
>
> If the function body is not on the same line, then it must come as an **indented** block of code which starts on a new line after `:` and has at least one line of code.

> **Caution**
>
> Indentations in constructs such as functions, classes, and loops are significant. This is in contrast to languages such as C/C++ which use curly braces `{}` instead of indentation.

```
def func(x):  # Not okay! The function body is missing.
```

```
SyntaxError: incomplete input (2172635986.py, line 1)
  Cell In[1], line 1
    def func(x):  # Not okay! The function body is missing.
                                                            ^
SyntaxError: incomplete input
```

> **Tip**
>
> You can use ellipsis `...` or `pass` in place of the function body. This is useful when e.g. the actual implementation is not ready yet and it is left for later.

```
def func(x): ...

def func(x): pass

def func(x):
    ...

def func(x):
    pass      # We will replace `pass` with the actual implementation later!
```

```
def multiply(x, y):
x * y  # Not okay! The function body must be indented.
```

```
IndentationError: expected an indented block after function definition on line 1 (4085265091.py, line 2)
  Cell In[3], line 2
    x * y  # Not okay! The function body must be indented.
    ^
IndentationError: expected an indented block after function definition on line 1
```

> **Tip**
>
> The actual number of spaces used for indentation does not matter. However, consistency matters. It is common to use two or four spaces as indentation.

```
# Both of the following are okay!

def multiply1(x, y):
    x * y  # 4 spaces

def multiply2(x, y):
  x * y  # 2 spaces
```

8.2 Functions implicitly return None

> **Caution**
>
> If the function body lacks an explicit `return` statement, then the function implicitly returns `None`.

```
# Both add1 and add2 return `None`

def add1(x, y):
    x + y

def add2(x, y):
    x + y
    return None

print(add1(1, 2))
print(add2(1, 2))
```

```
None
None
```

8.3 Parameters versus arguments

> **Tip**
>
> Variables which are part of the function signature are referred to as function **parameters**. The actual values which are passed to the function are called **arguments**. However, it is not uncommon to use them interchangeably. This could be okay as long as one knows the difference of the terms and it does not cause any confusion.

```
# `x` is a function parameter.
def double(x):
    return x * 2
```

```
# `3.5` is an argument. It replaces `x` when the function is called.
double(3.5)
```

```
7.0
```

> **Tip**
>
> A function can accept an undefined number of arguments. We use an asterisk `*` to pack all the passed arguments into a tuple (Section 6.2).

```
def func():
    print("I do not have any arguments!")

func()
```

```
I do not have any arguments!
```

```
def add_all(*args):    # All passed arguments are packed into a tuple.
    result = 1
    for x in args:     # Since `args` is a tuple we can iterate over it.
        result += x
    return result

add_all(2, 3, 5)
```

```
11
```

8.3.1 Keyword arguments

> **Tip**
>
> Functions can have parameters with default values. All parameters with default values must follow the positional parameters. When calling the function, arguments can be passed to the function both as positional or the so-called **keyword arguments**.

```
def add(x, y=1):     # `y` is optional here. If not provided it is assumed to be `1`.
    return x + y

print(add(2, 3))     # `y` is 3 due its position.

print(add(2, y=10))  # `y=10` is passed as a keyword argument.

print(add(6))        # Okay! `y=1` is implicitly assumed.
```

```
5
12
7
```

```
def add(y=1, x):   # Not okay! Positional parameters must come before optional ones.
    return x + y
```

```
SyntaxError: parameter without a default follows parameter with a default (4157070553.py, line 1)
  Cell In[11], line 1
    def add(y=1, x):   # Not okay! Positional parameters must come before optional ones.
                   ^
SyntaxError: parameter without a default follows parameter with a default
```

```
def add(x, y=1):     # The function definition is okay!
    return x + y

print(add(y=3, 1))   # Not okay. Positional arguments must come first.
```

```
SyntaxError: positional argument follows keyword argument (2200853574.py, line 4)
  Cell In[12], line 4
    print(add(y=3, 1))   # Not okay. Positional arguments must come first.
                     ^
SyntaxError: positional argument follows keyword argument
```

> **Caution**
>
> In case of multiple keyword arguments, their order does not matter if they are passed as keyword arguments. If they are passed as positional arguments the order matters.

```
def func(x, y=1, z=0):
    return x * y + z

print(func(5, y=4, z=3))  # The order for keyword arguments does not matter!
print(func(5, z=3, y=4))  # The order for keyword arguments does not matter!
```

```
23
23
```

```
def func(x, y=1, z=0):
    return x * y + z

print(func(5, 4, 3))  # The order matters, since we are using positional arguments!
print(func(5, 3, 4))  # The order matters, since we are using positional arguments!
```

```
23
19
```

> **Tip**
>
> Similar to positional arguments, a function can accept an undefined number of keyword arguments. We use double asterisks ** to pack all the passed keyword arguments into a dictionary (Section 7.2).

```
def print_all(title, **kwargs):
    print(title)
    for key, value in kwargs.items():
        print(key, value)

print_all("Items", apples="red", bananas="yellow")
```

```
Items
apples red
bananas yellow
```

> **Caution**
>
> The order of function parameters and arguments is
>
> 1. positional arguments which have been explicitly defined
> 2. other positional arguments
> 3. the explicit keyword arguments
> 4. other keyword arguments
>
> Example
>
> ```
> def func(x, y, *args, apples="red", **kwargs):
>     ...
> ```

8.4 Functions as objects

> **Tip**
>
> Since functions are objects, they can be e.g. aliased or passed to other functions as argument.

```
def add(x, y):
    return x + y

# We would like to have an alias for `add`.
plus = add

plus(1, 3) == add(1, 3)
```

```
True
```

```
# Given a function `func` and its input `x`, return the reciprocal of the function.
# The reciprocal of `func(x)` is `1/func(x)`.
def reciprocal(func, x):
    return 1 / func(x)


def quadratic(x):
    return x**2 - 5*x + 4


reciprocal(quadratic, 5)
```

```
0.25
```

> **Tip**
>
> Since functions are objects, they have attributes and you can access their attributes. One of the useful attributes is `__name__`, which gives you the name of the function as a string.

```
# A function that prints the name of the given function `func`
def print_function_name(func):
    print("The name of the function is: ", func.__name__)

def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

# Functions are objects, so they can be elements of a list.
for func in [add, multiply]:
    print_function_name(func)
```

```
The name of the function is:  add
The name of the function is:  multiply
```

```
def add(x, y):
    return x + y


def multiply(x, y):
    return x * y


def do_math(func, x, y):

    result = func(x, y)

    # func.__name__ gives the name of the function!
    func_name = func.__name__

    result_string = f"{func_name}({x}, {y}) = {result}" # Remember f-strings?

    print(result_string)


for func in [add, multiply]:
    do_math(func, 2, 3)
```

```
add(2, 3) = 5
multiply(2, 3) = 6
```

8.5 Recursive functions

> **Tip**
>
> Python supports **recursive functions**. You can use them to simplify the implementation of some algorithms.

```
# Recursive implementation for the `factorial` function.
# We assume that `n` is always positive!
def factorial(n):
    if n < 2:
        return n
    else:
        return factorial(n-1) * n  # Call the same factorial function recursively!

factorial(10)
```

```
3628800
```

> **Caution**
>
> For mutable objects, ensure that you do not modify them when passed as arguments to a function. Always make a deep copy of them if you want to modify the passed objects.

```
def append_one(lst):
    lst.append(1)     # We are modifying a mutable object!
    return lst

original_lst = [1, 2, 3]
output_list = append_one(original_lst)

print(output_list)    # Output list has `1` appended at the end. Good!
print(original_lst)   # Oops! We have accidentally changed the original list as well.
```

```
[1, 2, 3, 1]
[1, 2, 3, 1]
```

8.6 Putting it all together with an example

We present an example which reviews many of the useful features we have discussed so far. We aim to find the roots of functions using Newton's method, which is an iterative method.

Assume that you have a function $f(x)$ whose roots you want to find. We refer to derivate of this function as $f'(x)$, i.e.

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}$$

The Newton's method works as follows

1. **initial guess**: start with a guess for the root and name it x_0.
2. **value of the function**: compute the value of function at x_0, i.e.

$$f(x_0)$$

3. **value of the derivate**: compute the value of derivate at the same point, i.e.

$$f'(x_0)$$

4. **correction factor**: divide the result of step 2 by that of step 3, i.e.

$$\frac{f(x_0)}{f'(x_0)}$$

5. **update the guess**: subtract the result of step 4 from x_0 and name it x_1, i.e.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

6. **iteration**: go to step 2, replace x_0 with x_1 and repeat the process to get successive guess values, i.e.

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n$$

7. **convergence**: stop when you have reached a point where $f(x_n)$ is close enough to zero.

One can simplify the Newton's method as follows

$$x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$$

We implement a function which calculates the root. We name it `find_root`. The parameters of `find_root` are

- `func`: the input function for which we aim to find the root
- `derivative`: the derivate of the same input function
- `x_0`: initial guess for the root
- `threshold`: convergence threshold, which is by default set to `0.001`
- `max_iterations`: maximum number of iterations after which we stop regardless of whether we have reached convergence or not. This parameter has a default value of `100`.

```
def find_root(func, derivative, x_0, threshold=1e-3, max_iterations=100):
    for _ in range(max_iterations):
        correction_factor = func(x_0) / derivative(x_0)
        x_new = x_0 - correction_factor

        if abs(func(x_new)) < threshold:
            return x_new
        else:
            x_0 = x_new
```

Let's go through the function line by line:

- `def find_root(func, derivative, x_0, threshold=1e-3, max_iterations=100)` – The signature of the function according to our requirements. Note the first three arguments are positional and the last two are keyword arguments.
- `for _ in range(max_iterations)` – Iterate over a range of numbers from `0` to `max_iterations` (exclusive). Since we do not need the actual value of the iterator we have set it to `_`.
- `correction_factor = func(x_0) / derivative(x_0)` – Evaluate the input function and its derivate at `x_0` and calculate the correction factor.
- `x_new = x_0 - correction_factor` – Find a new value for the root.
- `if abs(func(x_new)) < threshold` – Compare the absolute value of the function at `x_new` and check whether it is below the threshold.
- `return x_new` – If `True`, return the value of `x_new` since we have reached convergence.
- `x_0 = x_new` – If `False`, update the guess and repeat the process.

> **Caution**
>
> If we do not reach convergence after `max_iterations`, we exit the `for` loop and return from the function. Since we have not specified an explicit value for this case, the function will implicitly return `None`.

Now let's test `find_root()`. We start with a case for which we know the root, so that we can verify the result. Assume the function is $f(x) = x^2 + 5x - 24$. We know from factorization that this function can be also written as $f(x) = (x - 3)(x + 8)$ which shows that the function has two roots, i.e. $x = 3$ and $x = -8$. We also know that the derivative of the function is $f'(x) = 2x + 5$.

```
def quadratic(x):
    return x**2 + 5*x - 24

def derivative(x):
    return 2*x + 5

# We start with `x=10` as our guess for the root
find_root(quadratic, derivative, 10, threshold=1e-6)
```

```
3.000000000000824
```

```
# Let's try with `x=-10` as our guess for the root
find_root(quadratic, derivative, -10, threshold=1e-6)
```

```
-8.000000000001084
```

Great! We got both roots by starting with different initial guesses.

Now let's consider a case for which the Newton's method does not work, e.g. $f(x) = x^2 + 1$ for which the derivative is $f'(x) = 2x$. This function does not have any real roots, it only has complex roots. As a result, Newton's method does not converge for this function.

```
def quadratic(x):
    return x**2 + 1

def derivative(x):
    return 2*x

# We start with `x=2` as our guess for the root
root = find_root(quadratic, derivative, 2, threshold=1e-1, max_iterations=100000)

print(root)
```

```
None
```

Note that we got `None` which means we could not converge and have reached the maximum number of iterations.

9 Classes

Classes are defined by the `class` keyword. The common convention is to use **CamelCase** for class names. However, there are exceptions to this convention for built-in classes (see Section 2.1.3), e.g. `int` is both a class and a type, and not a function.

> **Tip**
>
> Similar to functions we can leave the body of a class as `...` or `pass`.

```
class Course:
    pass

class Book:
    ...

class Animal: ...

class Fruit: pass
```

> **Note**
>
> A class is a bundle of methods (functions) and data that are closely related. In the language of object-oriented programming, this bundling is referred to as **encapsulation**.

> **Note**
>
> Classes are **blueprints** for objects, and objects are **instances** of classes. An actual object comes into existence when the class is instantiated. This is similar to having a house as an actual physical object in the real world, which is made out of e.g. concrete, and the blueprint of the same house which exists on a piece of paper.

```
class Course:
    pass

# `tnm119` is an object, and an instance of `Course`.
tnm119 = Course()
```

```
class Course:
    pass

tnm119 = Course()

# `tnm119` is of type `Course`.
type(tnm119) is Course
```

```
True
```

9.1 Instances and inheritance

> **Note**
>
> A class can inherit from other classes.

```
class Animal:
    pass

# `Cat` inherits from `Animal`.
# `Cat`` is a subclass of `Animal`.
# `Animal` is the superclass of `Cat`.
class Cat(Animal):
    pass
```

> **Tip**
>
> To check whether an object is an instance of a class, use `isinstance()` function.

> **Tip**
>
> To check whether a class is a subclass of another class, use `issubclass()` function.

```
class Course:
    pass

tnm119 = Course()

isinstance(tnm119, Course)
```

```
True
```

```
print(isinstance(2.0, float))
print(isinstance(2.0, int))
```

```
True
False
```

```
class Animal:
    pass

class Cat(Animal):
    pass


issubclass(Cat, Animal)
```

```
True
```

```
isinstance(Cat, Animal)
```

```
False
```

> **Note**
>
> An object is an instance of its direct class, as well as all of its super classes.

```
class Animal:
    pass

class Cat(Animal):
    pass


cat = Cat()

print(isinstance(cat, Cat))
print(isinstance(cat, Animal))
```

```
True
True
```

> **Caution**
>
> In Python, classes are objects themselves. They are all instances of the `type` class. A class whose instances are also classes is called a **metaclass**. An example of a metaclass is `type`.

```
type(Course)
```

```
type
```

```
isinstance(Course, type)    # `Course` is an instance of `type`.
```

```
True
```

```
isinstance(Course, object)  # `Course` is an instance of `object` as well.
```

```
True
```

```
issubclass(type, object)    # Since `type` itself is a subclass of `object`!
```

```
True
```

9.2 Instance methods

> **Note**
>
> **Instance methods**, or methods for short, are functions that are associated with objects. They can be defined in the same way as functions are defined. They are accessed by a dot (`.`) after the object, i.e. `<object>.<method>(<args>)`.

> **Note**
>
> All instance methods accept a first mandatory argument which is the object itself. This argument can be named anything, but it is conventionally named `self`.

> **Caution**
>
> There are exceptions to this. For examples, **static methods** do not have a first mandatory argument. Moreover, **class methods** have the class itself as the first argument and not the instance. We do not cover static and class methods in this refresher and only focus on **instance methods**.

```
class Course:
    def print_object_id(self):  # `self` is the mandatory first argument to all methods.
        object_id = id(self)
        print(object_id)

course = Course()

print(id(course))
course.print_object_id()
```

```
140348204075488
140348204075488
```

9.2.1 Magic methods

> **Note**
>
> Classes in Python can have a number of special methods, aka **magic methods**. These methods have specific use cases. They all start with and end in double underscores, e.g. `__init__`. As a result, they are also known as *dunder* methods, i.e. "d" from *double* and "under" from *underscore*, hence the term "dunder"!

> **Caution**
>
> Some magic methods have default implementations. i.e. when you create a class they are added to the class, right out of the box. Examples of magic methods with default implementations are `.__init__()` and `.__str__()`. However, the default implementations might not be necessarily what we want. In such cases, we can change and customize the behavior of these methods. This is referred to as **overriding**.

> **Caution**
>
> Other magic methods do not have default implementations. i.e. you have to explicitly define and implement them for every class that you create. Examples of such magic methods are `.__add__()` and `.__sub__()`.

9.2.1.1 `__init__()`

> **Tip**
>
> One of the commonly-used magic methods is `.__init__()`. It is essentially the constructor of the class. One can initialize object attributes (data) in the constructor.

```
# `.__init__()` has been added automatically, so that we can initialize the class.
class Course:
    pass

# Without `.__init__()` we could not make objects from `Course`.
tnm119 = Course()
print(tnm119)
```

```
<__main__.Course object at 0x7fa55cc38980>
```

```
class Course:
    def __init__(self, title):  # We are overriding the constructor!
        self.title = title      # We define a new object attribute and initialize it.

    def print_title(self):      # `self` is the mandatory first argument.
        print(f"The course title is: {self.title}")  # Remember f-strings?

course = Course("TNM119")
course.print_title()
```

```
The course title is: TNM119
```

9.2.1.2 `__str__()`

> **Tip**
>
> Another useful magic method is `.__str__()`. It is automatically called when one wants to convert the object into a string, such as when you want to use `print()`.

```
# `.__str__()` has been added automatically, so that we can print the object.
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

vector = Vector(1.5, 2.0)

# `print()` needs a string version of the `vector` so `.__str__()` is called!
# However, the return value of the default `.__str__()` is not that useful.
print(vector)
```

```
<__main__.Vector object at 0x7fa55cc380b0>
```

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # This method is called, whenever we need the string form of the object!
    # We override this method to return our desired string.
    def __str__(self):
        return f"Vector(x={self.x}, y={self.y})"   # We customize the string.

vector = Vector(1.5, 2.0)
print(vector)    # Isn't this better?
```

```
Vector(x=1.5, y=2.0)
```

9.2.1.3 __add__()

> **Tip**
>
> Other useful magic methods allow us to add support for arithmetic operations to objects. Examples of such magic methods are `.__add__()` for addition, and `.__sub__()` for subtraction.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Vector(x={self.x}, y={self.y})"   # We customize the string.

v1 = Vector(1.5, 2.0)
v2 = Vector(2.0, 3.0)


# When we use `+`, Python implicitly calls `.__add__()`.
# Note that `.__add__()` does not have a default implementation.
# As a result, addition is not defined/supported for `Vector` yet!
# The following will raise an exception.
v1 + v2
```

```
TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 17
     10 v2 = Vector(2.0, 3.0)
     13 # When we use `+`, Python implicitly calls `.__add__()`.
     14 # Note that `.__add__()` does not have a default implementation.
     15 # As a result, addition is not defined/supported for `Vector` yet!
     16 # The following will raise an exception.
---> 17 v1 + v2
TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'
```

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # We are not overriding this, as it does not have a default implementation.
    # We are implementing it ourselves.
    def __add__(self, other):  # We assume that `other` is also of type `Vector`
        # It is up to us to define what `addition` means!
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector(x={self.x}, y={self.y})"

v1 = Vector(1.5, 2.0)
v2 = Vector(2.0, 3.0)

v = v1 + v2
print(v)    # Interesting!
```

```
Vector(x=3.5, y=5.0)
```

9.2.1.4 List of magic methods

> **Tip**
>
> To get the list of all available magic methods for a specific class, use `dir()` function.

```
dir(Vector)
```

```
['__add__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
```

```
 '__subclasshook__',
 '__weakref__']
```

Note that in the example above since we have implemented `__add__`, it shows up in the list. However, other magic methods, e.g `__sub__` for subtraction is not in the list, since it does not have a default implementation and we have not provided one.

Compare the example above with the `int` class, which supports all arithmetic operations,

```
dir(int)
```

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__rand__',
 '__rdivmod__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rfloordiv__',
 '__rlshift__',
 '__rmod__',
 '__rmul__',
 '__ror__',
```

```
'__round__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__trunc__',
'__xor__',
'as_integer_ratio',
'bit_count',
'bit_length',
'conjugate',
'denominator',
'from_bytes',
'imag',
'is_integer',
'numerator',
'real',
'to_bytes']
```

10 Libraries

10.1 Structure of libraries

When talking about libraries in Python, we have the following concepts which are related

- **module**: a single Python file ending in `.py`
- **package**: a group (directory) of modules, which also includes an `__init__.py` file
- **sub-package**: a package within another package
- **library**: a group of packages or modules

Below is an example of the structure of a library in Python

```
- library
    - __init__.py

    - module.py

    - package_one
        - __init__.py
        - module_one_1.py
        - module_one_2.py

        - sub_package
            - __init__.py
            - module_sub.py

    - package_two
        - __init__.py
        - module_two_1.py
        - module_two_2.py
```

> **Caution**
>
> All packages include an `__init__.py` file. This file can even be empty. It needs to be there so that Python knows to treat the directory as a package[1].

10.2 Third-party libraries

Python has a rich built-in library, called Python Standard Library which is part of the Python installation. However, like many other programming languages one cannot get much far by relying only on the standard library.

Third-party libraries are needed to e.g. augment the language features, improve the performance. Python has a wealth of third-party libraries, some of which are considered as the *standard* tool for accomplishing various tasks in Python.

[1] This is only true about regular packages. Namespace packages in Python do not need to have the `__init__.py` file. However, this is beyond the scope of the refresher.

> **Note**
>
> Among many of the third-party libraries, the following are useful for machine learning and data science
>
> - NumPy
>   large multi-dimensional arrays
> - Matplotlib
>   visualization
> - SciPy
>   functions and utilities used in science and engineering
> - pandas
>   datasets
> - scikit-learn
>   machine learning

In this refresher, we only cover NumPy and Matplotlib.

10.3 How to use libraries

To use a library you need to first install it and then import into your code/script.

10.3.1 Installing libraries

> **Note**
>
> Assuming that you are using `conda`, you can install a library via `conda install <library-name>`.
> Example:
> The following installs NumPy

```
conda install numpy
```

10.3.2 Importing libraries

To import a library you have several options. We use NumPy as an example. However, all of the following also apply to any other library.

10.3.2.1 Import by the library name only

```
# Import the library.
import numpy

print(numpy.__version__)
```

```
2.2.1
```

10.3.2.2 Import by an alias for the library

```
# Import the library and assign an alias to it.
# The alias is usually an abbreviation of the original name.
# If you do this, the original name is no longer available in the namespace.
import numpy as np

print(np.__version__)
```

```
2.2.1
```

```
# NumPy is not available since we have imported it with an alias, i.e. `np`.
print(numpy.__version__)
```

```
NameError: name 'numpy' is not defined
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 # NumPy is not available since we have imported it with an alias, i.e. `np`.
----> 2 print(numpy.__version__)
NameError: name 'numpy' is not defined
```

10.3.2.3 Import individual entities from the library

```
# Import only what you need.
from numpy import sin, pi

print(pi)
print(sin)
```

```
3.141592653589793
<ufunc 'sin'>
```

10.3.2.4 Import individual entities with an alias

```
# Import only what you need with an alias.
from numpy import pi as PI

print(PI)
```

```
3.141592653589793
```

10.3.2.5 Import all the entities

```
# Import everything using `*`.
from numpy import *

# We did not import `pi` explicitly.
# When we use `*`, everything including `pi` will be imported.
print(pi)
```

```
3.141592653589793
```

> **Tip**
>
> Among all the aforementioned methods, we usually prefer to import a library with an alias, e.g. `import numpy as np`. Using aliases helps with readability and avoids name conflicts in the namespace. However, this might not be always desired. In such cases, we need to be careful about potential namespace conflicts.

10.4 Namespace conflicts

Namespace conflicts occur when we want to import several objects from different modules, and they all share the same name. In such cases, with each import, the object gets replaced by another object with the same name.

> **Caution**
>
> It is highly recommended that you avoid importing everything with `from numpy import *`. This can lead to name conflicts in the namespace and unexpected behavior. See the examples below.

```
import math


# We define a `sin()` function whose arguments are in degrees.
def sin(theta_degree):
    theta_radian = theta_degree * math.pi / 180
    return math.sin(theta_radian)

# We confirm that `sin(90 deg)` is `1.0`. Great!
print(sin(90))

# We do not even know what we have imported!
from numpy import *

# Why is it not `1.0`? I thought we have defined a `sin()` function for degrees.
# Oh! The `sin()` function has been replaced by NumPy `sin()` whose arguments are in radians!
print(sin(90))
```

```
1.0
0.8939966636005579
```

> **Caution**
>
> When importing individual entities from a library, ensure that it does not lead to any name conflicts in the namespace.

As an example, Python has a built-in module named `math` which has some functions with the same names as NumPy.

```
# NumPy has a `sin()` function.
from numpy import sin
print(sin)

# Python also has a built-in math module and it too has `sin()`.
from math import sin
print(sin)
```

```
<ufunc 'sin'>
<built-in function sin>
```

Now consider the following scenario

```
from numpy import pi, sin

# NumPy supports mathematical operations on lists.
print(sin([-pi, 0, pi]))
```

```
[-1.2246468e-16  0.0000000e+00  1.2246468e-16]
```

```
# some lines of code
# ...
# We forgot that we have imported `sin()` from NumPy earlier!
from math import pi, sin
# some more lines of code
# ...

# Oops! This does not work. Python built-in `math.sin()` does not work on lists.
# NumPy `sin()` and `pi` have been replaced by `math.sin()` and `math.pi`.
print(sin([-pi, 0, pi]))
```

```
TypeError: must be real number, not list
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[11], line 10
      4 from math import pi, sin
      5 # some more lines of code
      6 # ...
      7
      8 # Oops! This does not work. Python built-in `math.sin()` does not work on lists.
      9 # NumPy `sin()` and `pi` have been replaced by `math.sin()` and `math.pi`.
---> 10 print(sin([-pi, 0, pi]))
TypeError: must be real number, not list
```

Now look how things become more readable when we use aliasing,

```
import math
import numpy as np

# We have access to both functions at the same time!
# No name conflicts anymore. Great!
print(np.sin)
print(math.sin)
```

```
<ufunc 'sin'>
<built-in function sin>
```

11 Numpy

Python collections are not suitable for mathematical operations. For example, suppose you store an array of numbers in a Python list and you want to perform simple arithmetic operations on the list. Such operations can be very slow and you need to write several or complex lines of code for a simple task. Imagine the hassle when you need to perform complex matrix operations on Python lists!

First, note that when applied on Python lists, + and * do not even yield results that are expected from basic arithmetic. See lists (Section 6.1) for more information on this, but as an example see below. In the examples, + and * do not lead to element-wise addition or multiplication of the lists.

```
lst1 = [1, 2]
lst2 = [3, 4]

# This is not what we want! `+` concatenates the two lists.
# However, we want element-wise addition, i.e. `[4, 6]`
lst1 + lst2
```

```
[1, 2, 3, 4]
```

```
# This is not what we want! `* 2` duplicates the elements.
# However, we want multiplication of elements by `2`, i.e. `[2, 4, 6]`
[1, 2, 3] * 2
```

```
[1, 2, 3, 1, 2, 3]
```

This means we have to use iteration over lists, for a simple arithmetic operation. However, this is not the only disadvantage. Let's proceed and implement the simple addition of two lists in a mathematical sense.

```
import random
import time


# Example function to generate two lists of random floats and add them.
# We also time the function.
# `N` is the number of elements in each list.
def example(N):
    t1 = time.perf_counter()  # The time when the mathematical operations start.

    # two lists of random floats between 0 and 1.
    lst1 = [random.uniform(0, 1) for _ in range(N)]
    lst2 = [random.uniform(0, 1) for _ in range(N)]

    # `lst1 + ls2` leads to concatenation of the two lists. This is not what we want.
    # We have no choice but to implement the addition of two lists by using iteration.
    lst = [i+j for i, j in zip(lst1, lst2)]

    t2 = time.perf_counter()  # The time when the mathematical operations finish.

    elapsed_time = t2 - t1    # The duration of mathematical operations.
    return elapsed_time


elapsed_time_list = example(5_000_000)
print(f"{round(elapsed_time_list, 2)} seconds")
```

```
1.57 seconds
```

Close to two seconds for doing this simple operation only once! This is too slow!

NumPy has been conceived as the solution to such issues. It is the library for numerical operations on large and multi-dimensional arrays and matrices. It is based on C and therefore much faster than python built-in solutions such as using lists. Let's see the same example but this time using NumPy arrays!

```
import numpy as np
import time

# The same example but with NumPy.
def example(N):
    t1 = time.perf_counter()

    # two arrays of random floats between 0 and 1.
    arr1 = np.random.uniform(0, 1, N)
    arr2 = np.random.uniform(0, 1, N)
    # addition of two arrays with `+` is supported, right out the box!
    arr = arr1 + arr2

    t2 = time.perf_counter()

    elapsed_time = t2 - t1
    return elapsed_time

elapsed_time_numpy = example(5_000_000)
print(f"{round(elapsed_time_numpy, 2)} seconds")
```

```
0.06 seconds
```

And look at the performance boost now

```
boost = int(elapsed_time_list / elapsed_time_numpy)

print(f"Using NumPy, we achieved a boost of ~{boost} folds in performance!")
```

```
Using NumPy, we achieved a boost of ~26 folds in performance!
```

In addition to such a significant performance boost, see the comparisons below to note how simpler and more intuitive the NumPy expressions are.

> Generating an array of random numbers
>
> Python list
> `lst = [random.uniform(0, 1) for _ in range(N)]`
>
> NumPy array: shorter and more intuitive
> `arr = np.random.uniform(0, 1, N)`

> Addition of two arrays
>
> Python list
> `lst = [i+j for i, j in zip(lst1, lst2)]`
>
> NumPy array: supports addition of arrays, elegant!
> `arr = arr1 + arr2`.

11.1 Examples

11.1.1 Conversion of Python lists to NumPy arrays

```
import numpy as np

lst = [1, 2, 3]
arr = np.array(lst)

print(arr)
print(type(arr))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

11.1.2 Generating multi-dimensional arrays from nested lists

```
lst = [[1, 2, 3], [4, 5, 6]]
arr = np.array(lst)

arr
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

11.1.3 Getting the shape of the array

```
# An array with two dimensions, i.e. two rows and three columns, i.e. 6 elements in total.
arr.shape
```

```
(2, 3)
```

11.1.4 Getting the number of all elements along all axes

```
arr.size
```

```
6
```

11.1.5 Reshaping arrays

> **Note**
>
> Since NumPy arrays are multi-dimensional, they can be reshaped into other arrays with the same size. Let's see a few examples.

```
# Original array, 2x3
print(arr)
print(arr.shape)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

```
# Reshape into 3x2
arr_three_by_two = arr.reshape((3, 2))
print(arr_three_by_two)
print(arr_three_by_two.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
(3, 2)
```

```
# Reshape into 1x6, i.e. a single row
arr_row = arr.reshape((1, 6))
print(arr_row)
print(arr_row.shape)
```

```
[[1 2 3 4 5 6]]
(1, 6)
```

```
# Reshape into 6x1, i.e. a single column
arr_column = arr.reshape((6,1))
print(arr_column)
print(arr_column.shape)
```

```
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
(6, 1)
```

```
# Cannot reshape 6 elements into 2x4
arr_wrong_size = arr.reshape((2, 4))
```

```
ValueError: cannot reshape array of size 6 into shape (2,4)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[14], line 2
      1 # Cannot reshape 6 elements into 2x4
----> 2 arr_wrong_size = arr.reshape((2, 4))
ValueError: cannot reshape array of size 6 into shape (2,4)
```

11.1.6 Flattening arrays

> **Tip**
>
> Numpy arrays can be converted into one-dimensional arrays, using both `.reshape()` or `.flatten()`.

```
# Reshape into a flat array (one-dimensional) with 6 elements
arr_flat = arr.reshape(6,)
print(arr_flat)
print(arr_flat.shape)
```

```
[1 2 3 4 5 6]
(6,)
```

```
# Flatten array using the `.flatten()` method
arr_flat = arr.flatten()
print(arr_flat)
print(arr_flat.shape)
```

```
[1 2 3 4 5 6]
(6,)
```

11.1.7 Stacking arrays

We can stack NumPy arrays, e.g. horizontally or vertically. Stacking is similar to concatenation of lists, but we have control over which axis the stacking happens.

> **Tip**
>
> The functions that we use for stacking are `dstack()`, `hstack()`, `vstack()`, and `stack()`.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(arr1)
print(arr2)
```

```
[1 2 3]
[4 5 6]
```

```
#Vertical stacking, along axis=0
np.vstack((arr1, arr2))
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
#Horizontal stacking, along axis=1
np.hstack((arr1, arr2))
```

```
array([1, 2, 3, 4, 5, 6])
```

```
#Depth-wise stacking
np.dstack((arr1, arr2))
```

```
array([[[1, 4],
        [2, 5],
        [3, 6]]])
```

> **Tip**
>
> `stack()` allows us to specify the axis along which we want to stack arrays. This is especially useful when we have many dimensions.

```
# Equivalent to vstack
np.stack((arr1, arr2), 0)
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

11.1.8 NumPy arrays are multi-dimensional

```
arr = np.array(
    [
        [
            [1, 2, -1, -2], [3, 4, -3, -4], [5, 6, -5, -6]
        ],
        [
            [7, 8, -7, -8], [9, 10, -9, -10], [11, 12, -11, -12]
        ]
    ]
)

arr.shape # two rows, three columns, four elements along the third axis (depth)
```

```
(2, 3, 4)
```

```
# Get the number of dimensions
np.ndim(arr)
```

```
3
```

11.1.9 Indexing elements across different axes

```
arr[1, 2, 3]   # Element at row: 0, column: 1, and depth: 2 --> -12
```

```
np.int64(-12)
```

```
print(arr[0,1,:])  # All elements of the zeroth row and the first column
```

```
[ 3  4 -3 -4]
```

```
print(arr[1,:,:])   # All elements of the first row
```

```
[[  7   8  -7  -8]
 [  9  10  -9 -10]
 [ 11  12 -11 -12]]
```

```
print(arr[:,:,-1])   # Last elements along the last (depth) axis
```

```
[[ -2  -4  -6]
 [ -8 -10 -12]]
```

11.1.10 Filtering arrays

> **Tip**
>
> You can use boolean expressions (e.g. comparisons) alongside indexing to filter NumPy arrays.

> **Warning**
>
> When using boolean expressions with indexing, the individual conditions can be combined with `&` for a logical *and* and `|` for a logical *or*. You cannot use Python built-in `and` and `or`.

```
positive_and_even = arr[(arr > 0) & (arr % 2 ==0) ]
positive_and_even
```

```
array([ 2,  4,  6,  8, 10, 12])
```

> **Tip**
>
> In addition, you can use `np.where()` to achieve the same thing and receive the indices where the conditions are `True`.

```
indices_where_condition_is_true = np.where((arr > 0) & (arr % 2 == 0))
arr[indices_where_condition_is_true]
```

```
array([ 2,  4,  6,  8, 10, 12])
```

> **Tip**
>
> NumPy operations can be done on different axes depending on our use case.

```
# Sum of all elements in the array.
arr.sum()
```

```
np.int64(0)
```

```
# Sum along the row axis.
arr.sum(axis=0)
```

```
array([[  8,  10,  -8, -10],
       [ 12,  14, -12, -14],
       [ 16,  18, -16, -18]])
```

```
# Sum along the column axis.
arr.sum(axis=1)
```

```
array([[  9,  12,  -9, -12],
       [ 27,  30, -27, -30]])
```

```
# Sum along the depth axis.
arr.sum(axis=2)
```

```
array([[0, 0, 0],
       [0, 0, 0]])
```

11.1.11 Initializing arrays

> **Note**
>
> NumPy offers some convenience functions to initialize arrays.

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((2, 3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

```
arr = np.empty((2, 3))
arr.fill(15)

arr
```

```
array([[15., 15., 15.],
       [15., 15., 15.]])
```

```
arr = np.zeros((4, 4))
np.fill_diagonal(arr, 3)

arr
```

```
array([[3., 0., 0., 0.],
       [0., 3., 0., 0.],
       [0., 0., 3., 0.],
       [0., 0., 0., 3.]])
```

```
# Identity
np.eye(4)
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

11.1.12 NumPy types

> **Note**
>
> NumPy arrays have (numeric) data types similar to what we have in the `C` language. This is different from the Python built-in types. However, Python and NumPy types are implicitly converted to each other. To get the type of a Numpy array use the `.dtype` property. For example `np.int64` corresponds to a 8-byte integer, i.e. 64 bits are used to store values of type `np.int64`.

```
arr.dtype
```

```
dtype('float64')
```

> **Tip**
>
> You can convert between types with `.astype()`. You can also specify the type explicitly when making a new array.

```
arr_int = np.ones((2, 3), dtype=np.int64)
print(arr_int.dtype)

arr_float = np.ones((2, 3), dtype=np.float64)
print(arr_float.dtype)
```

```
int64
float64
```

```
arr = np.array([1, 2, 3])
print(arr.dtype)

# Convert the type, this is an example of type promotion (int to float).
arr_float = arr.astype(np.float64)
print(arr_float.dtype)
```

```
int64
float64
```

11.2 Arithmetic and vector operations

NumPy arrays support both arithmetic and vector operations as we will see in the examples below. For the operations to work, the shapes of the arrays must be compatible according to the underlying mathematical rules.

> **Note**
>
> For simple arithmetic operations, when the two arrays have the same shape, the operation is performed element-wise. More specifically, the operator acts on each element from the first array and its counterpart from the second array as a pair.

Addition of two arrays

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition of elements
# i.e. [1+4, 2+4, 3+6]
arr1 + arr2
```

```
array([5, 7, 9])
```

```
# Element-wise multiplication of elements
# i.e. [1*4, 2*4, 3*6]
arr1 * arr2
```

```
array([ 4, 10, 18])
```

```
# This does not work as shapes are not compatible.
# How can we add an array of size 3 to another array of size 2?
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5])

arr1 + arr2
```

```
ValueError: operands could not be broadcast together with shapes (3,) (2,)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[44], line 6
      3 arr1 = np.array([1, 2, 3])
      4 arr2 = np.array([4, 5])
----> 6 arr1 + arr2
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Addition of a scalar to an array

```
arr + 2.0
```

```
array([3., 4., 5.])
```

Multiplication of an array by a scalar

```
arr * 2
```

```
array([2, 4, 6])
```

> **Note**
>
> For vector products we have a number of different options, e.g. dot products and cross products. In this refresher, we only cover the dot product, which can be performed using `np.dot()`.

> **Note**
>
> Dot product of two one-dimensional arrays a and b, is mathematically defined as
>
> $$a.b = \sum_{i=0}^{N} a_i.b_i$$
>
> where N is the size of both arrays.
>
> Example:
>
> $$a = [1, -2, 3]$$
> $$b = [4, 1, 2]$$
> $$a.b = (1 \times 4) + (-2 \times 1) + (3 \times 2) = 8$$

```
a = np.array([1, -2, 3])
b = np.array([4, 1, 2])
np.dot(a, b)
```

```
np.int64(8)
```

> **Tip**
>
> NumPy supports some operations which are common in linear algebra, e.g. determinant and inverse.

```
# A `2 X 2` array (matrix)
arr = np.array([[1, 2], [3, 4]])

# determinant
print("The determinant of the array is: ", np.linalg.det(arr))

#inverse
print("The inverse of the array is:\n", np.linalg.inv(arr))
```

```
The determinant of the array is:  -2.0000000000000004
The inverse of the array is:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

11.3 Comparison of arrays

Since NumPy arrays can include more than one element and are multi-dimensional, Numpy offers a number of different comparison functions to support different use cases.

```
arr1 = np.array([3, 4], dtype=np.int64)
arr2 = np.array([3, 4], dtype=np.float64)

# Two arrays have the same shape and elements
print(np.array_equal(arr1, arr2))

print("---")

# Element-wise comparison of the same thing
print(np.equal(arr1, arr2))
```

```
True
---
[ True  True]
```

```
arr1 = np.array([3, 4])
arr2 = np.array([3.01, 4.01])

# All elements in the two arrays are close given an absolute tolerance of `1e-2`
print(np.allclose(arr1, arr2, atol=1e-2))

print("---")

# Element-wise comparison of the same thing
print(np.isclose(arr1, arr2, atol=1e-2))
```

```
True
---
[ True  True]
```

```
arr1 = np.array([10, 4])
arr2 = np.array([3, 5])

# Regular comparison operators are element-wise.
arr1 < arr2
```

```
array([False,  True])
```

11.4 Infinity (inf) and Not a Number (nan)

These are two special objects to cover edge cases of 1/0 (`np.inf`) and 0/0 (`np.nan`).

> **Warning**
>
> When such edge cases occur as a part of NumPy operations, NumPy does not raise an exception. Instead it issues a warning.

> **Warning**
>
> It is not recommended to suppress warnings in general. Usually such warnings indicate an issue with the code and/or the data.

If you really want to suppress warnings, you can do it via

```
import warnings
warnings.filterwarnings("ignore")
```

```
# Python is performing the division, not NumPy! It will raise an exception.
1 / 0
```

```
ZeroDivisionError: division by zero
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[53], line 2
      1 # Python is performing the division, not NumPy! It will raise an exception.
----> 2 1 / 0
ZeroDivisionError: division by zero
```

```
# NumPy handles this. No exceptions will be raised.
# NumPy will issue warnings that we suppressed earlier.

np.array([1]) / np.array([0])
```

```
array([inf])
```

```
arr1 = np.array([1, 3, 3, 0, 5, 0])
arr2 = np.array([0, 2, -4, 0, 5, 6])

arr1 / arr2
```

```
array([  inf,  1.5 , -0.75,   nan,  1.  ,  0.  ])
```

> **Warning**
>
> Since `np.nan` is not a number, its comparison to any other value yields `False`, even itself! This is, however, different for `np.inf` which behaves like a number.

```
# nan is not equal to anything, not even itself!
np.nan == np.nan
```

```
False
```

```
# np.inf behaves like a number
print(np.inf == np.inf)
print(np.inf > 10)
print(np.inf < np.inf)
print(-1 * np.inf)
print(np.inf / np.inf)
```

```
True
True
False
-inf
nan
```

> **Tip**
>
> NumPy functions `isnan`, `isinf`, and `isfinite` are useful to handle edge cases.

```
arr = np.array([np.inf, 1, np.nan, 3])

np.isnan(arr)
```

```
array([False, False,  True, False])
```

```
np.isinf(arr)
```

```
array([ True, False, False, False])
```

```
np.isfinite(arr)
```

```
array([False,  True, False,  True])
```

11.5 Other useful functions

11.5.1 Generating an array given a range and the step

Generating an array of floats equally spaced in $[-2 \times \pi, 2 \times \pi)$ in steps of 0.01

```
np.arange(-2*np.pi, 2*np.pi, 0.01)
```

```
array([-6.28318531, -6.27318531, -6.26318531, ...,  6.25681469,
        6.26681469,  6.27681469], shape=(1257,))
```

11.5.2 Generating an array given a range and the number of elements

Generating an array of 2000 floats equally spaced in $[-2 \times \pi, 2 \times \pi)$

```
np.linspace(-2*np.pi, 2*np.pi, 2000)
```

```
array([-6.28318531, -6.27689898, -6.27061265, ...,  6.27061265,
        6.27689898,  6.28318531], shape=(2000,))
```

11.5.3 Random numbers from a normal distribution

```
# `10_000`` random floats from a normal distribution.
# The mean of the distribution is `2.0`.
# The standard deviation of the distribution is `0.1`.
np.random.normal(2.0, 0.1, 10_000)
```

```
array([2.01842505, 1.95329077, 1.96919234, ..., 1.9270683 , 2.09605467,
       2.07824076], shape=(10000,))
```

11.5.4 Random integers from a uniform distribution

```
# `10_000`` random integers from a uniform distribution between `10` and `20`.
# `20` is exclusive.
np.random.randint(10, 20, 10_000)
```

```
array([17, 19, 13, ..., 14, 18, 15], shape=(10000,))
```

11.5.5 Generating a mesh grid of independent variables to functions

This is useful when we have two arrays `x` and `y` as independent variables and another value, e.g. `z`, which is a function of both, i.e. $z = f(x, y)$. We then want to make a contour plot for `z`. In such a case we want all the combinations of `x` and `y` values. In other words, we need a two-dimensional meshgrid which includes all combinations of `x` and `y` values.

As an example suppose we have `x=[1, 2]` and `y=[-1, -2]`. To make a contour plot for `z`, we need four combinations of `[1, -1]`, `[1, -2]`, `[2, -1]`, `[2, -2]`. If we want to describe this combination as two arrays for `x` and `y`, one possibility is

```
xx = [
    [1, 2]
    [1, 2]
]
```

and

```
yy = [
    [-1, -1]
    [-2, -2]
]
```

Note that each element from `xx` when matches with its counterpart from `yy` gives one of the four desired combinations. For example `xx[0, 1]=1` has a counterpart of `yy[0, 1]=-2`, which gives `[1, -2]`.

For such cases, we can use the Numpy `meshgrid()` function.

```
x = np.array([1, 2])
y = np.array([-1, -2])

xx, yy = np.meshgrid(x, y)

print(xx)
print(yy)
```

```
[[1 2]
 [1 2]]
[[-1 -1]
 [-2 -2]]
```

See Chapter 12 for an example of a contour plot, where we utilize `meshgrid()`.

12 Matplotlib

Matplotlib is a user-friendly Python library for visualization. One can create two- and three-dimensional plots with minimum effort. In particular, Matplotlib has a module Pyplot which has been designed to have an interface similar to MATLAB. As a result, those with prior experience in plotting with MATLAB, find Pyplot familiar.

> **Tip**
>
> We mention a few examples in this chapter. However, for more examples, see the gallery of visualizations created by Matplotlib.

> **Note**
>
> In the following examples, when rendering the plots we have used LaTeX to typeset axes labels, legends,and the plot titles. This way we could show mathematical symbols and expressions such as θ and x^2. However, for your convenience, all the strings in the example scripts given below, do not include any LaTeX commands.
>
> If you like, you can render your plots so that axes labels, legends, and plot titles resemble the rendered plots below. For this purpose you need to
>
> 1. Install LaTeX from https://www.latex-project.org/
> 2. Change the regular strings to raw strings which include LaTeX mathematical expressions
>
> Example:
> Suppose you add a title to your plot with

```
plt.plot("sin(t)^2")
```

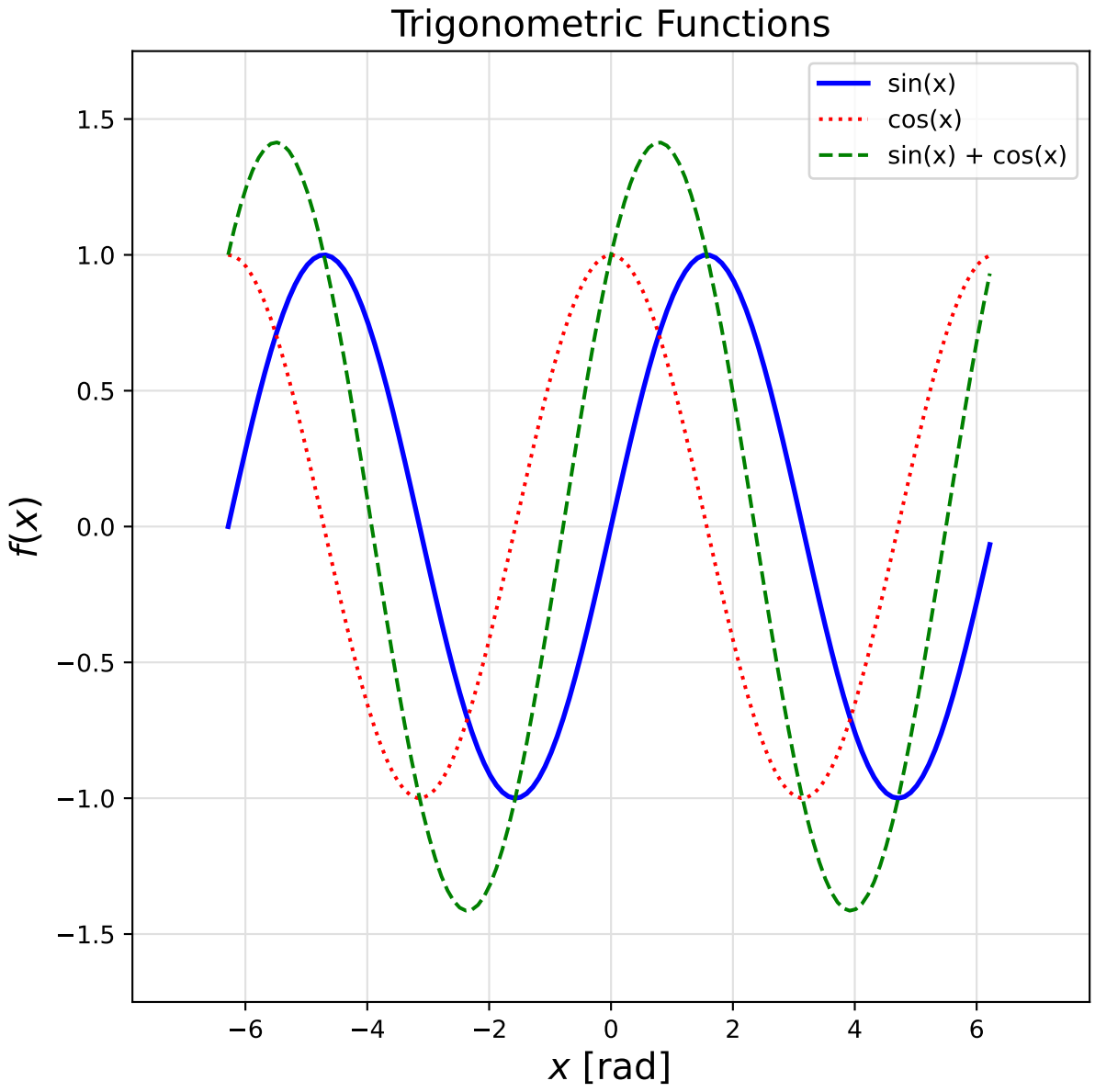
Now, if you change the title string to the following

```
plt.plot(r"$(\sin{\theta})^2$")
```

Then it will be rendered as $(\sin\theta)^2$ when showing the plot.

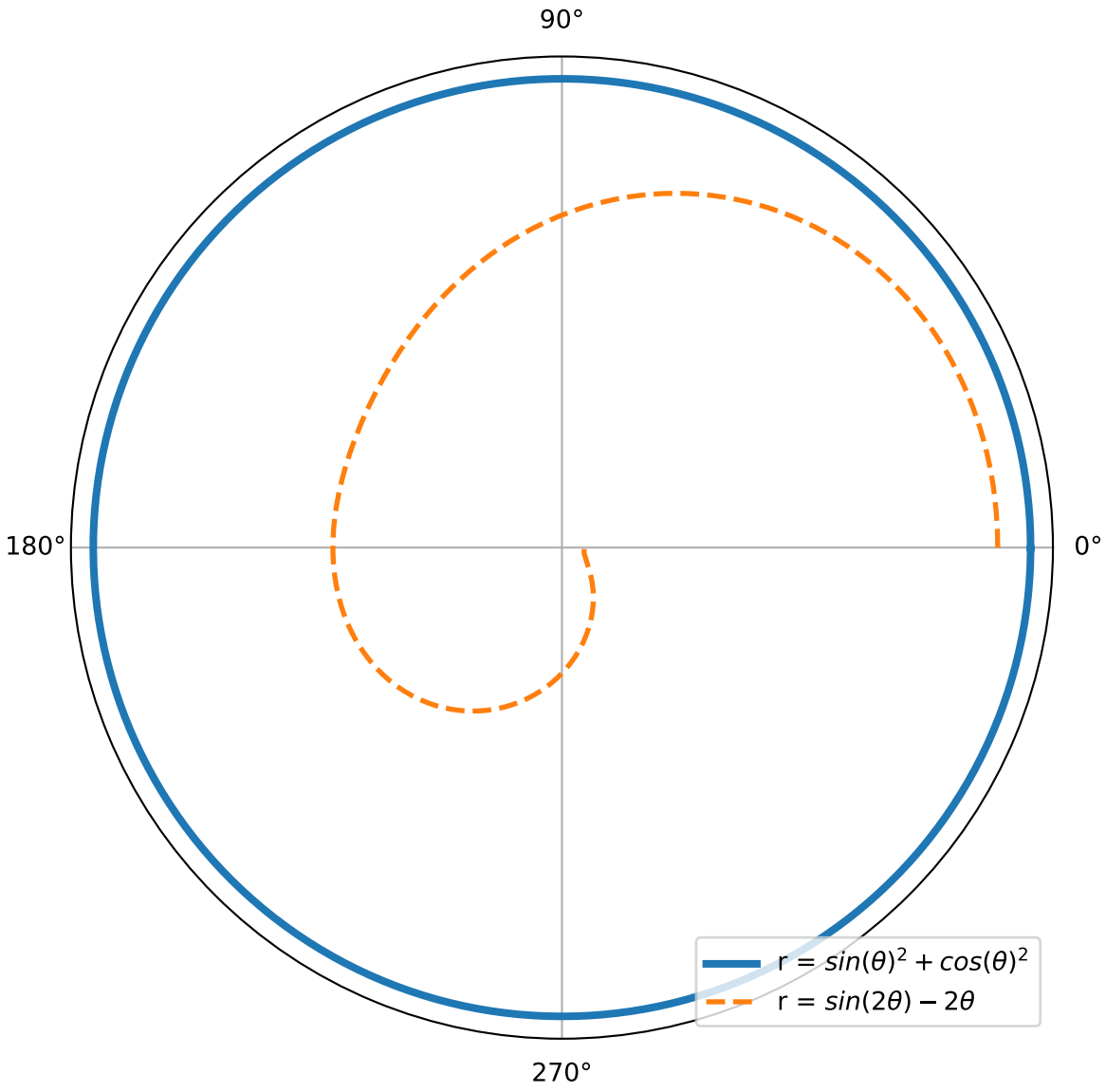
12.1 Example of a line plot

```
import numpy as np
import matplotlib.pyplot as plt

# Make a figure with the given size in inches.
figure, axes = plt.subplots(figsize=(7, 7))

# The independent variable (horizontal axis).
x = np.arange(-2*np.pi, 2*np.pi, 0.1)

y1 = np.sin(x)
y2 = np.cos(x)
y3 = y1 + y2

# Plot `x` versus `y1` in solid (-) blue line (b), with a line-width of 2 (lw).
# We also make a label for the curve which shows up in the legends.
plt.plot(x, y1, "-b", lw=2, label="sin(x)")

# Plot `x` versus `y2` in dotted (:) red line (r).
plt.plot(x, y2, ":r", label="cos(x)")

# Plot `x` versus `y3` in dashed (--) green line (g).
plt.plot(x, y3, "--g", label="sin(x) + cos(x)")

# Set the label (and its font size) for the horizontal axis.
plt.xlabel(r"x [rad]", fontsize=15)

# Set the label (and its font size) for the vertical axis.
plt.ylabel("f(x)", fontsize=15)

# Set the extent of the horizontal and vertical axexs.
plt.xlim([-2.5*np.pi, 2.5*np.pi])
plt.ylim([-1.75, 1.75])

# Add the legend using the labels that we have defined above.
plt.legend()

# Add a grid with a hexadecimal color of `#E0E0E0`.
# This color corresponds to RGB(224, 224, 224).
plt.grid(color="#E0E0E0")

# Add a title to the plot.
plt.title("Trigonometric Functions", fontsize=15)

# Render the plot.
plt.show()
```


Trigonometric Functions
sin(x)
cos(x)
sin(x) + cos(x)
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
f(x)
−6
−4
−2
0
2
4
6
x [rad]

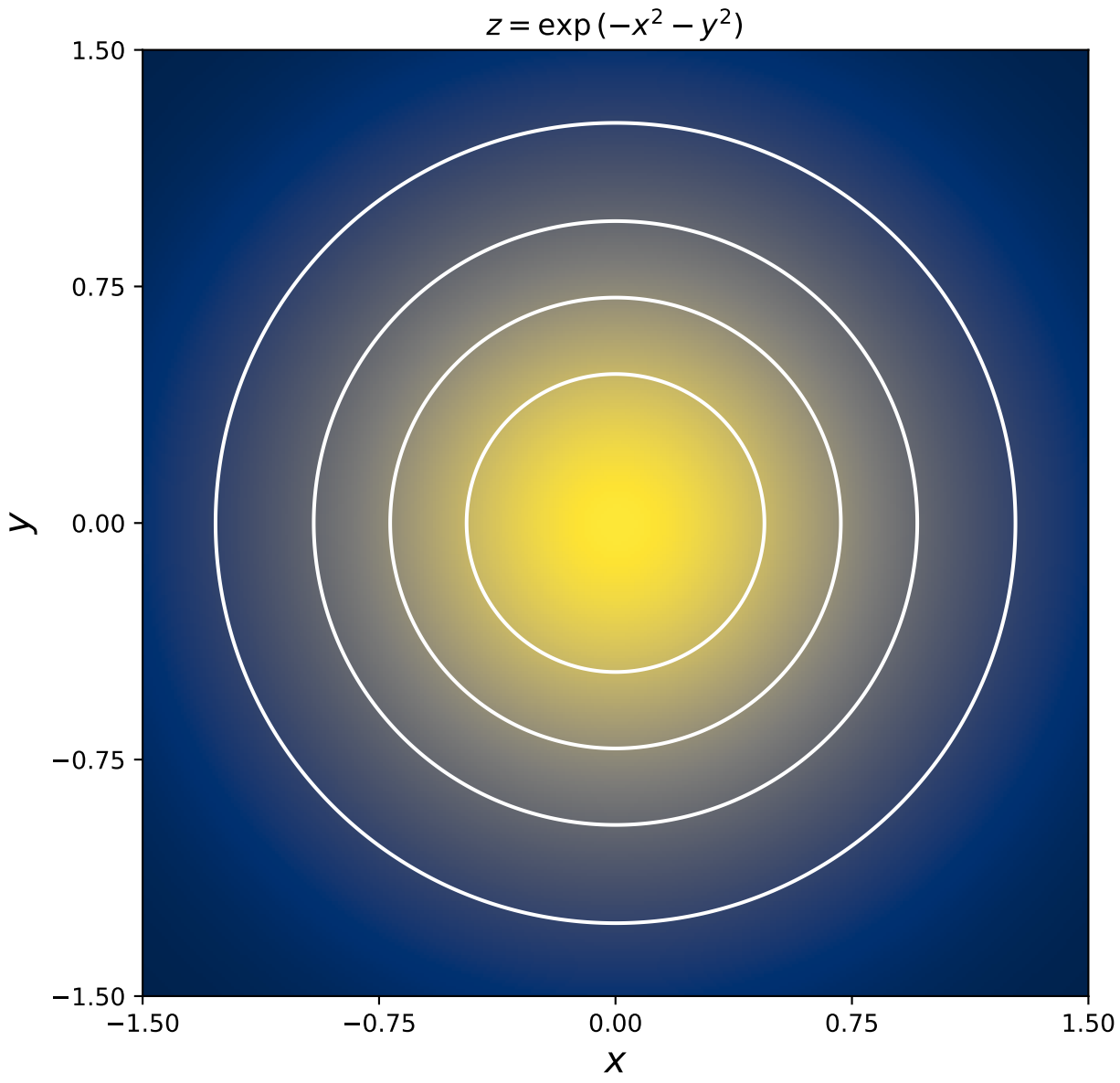
12.2 Example of a polar plot

```
import numpy as np
import matplotlib.pyplot as plt

# We set the projection to `polar`.
fig, ax = plt.subplots(subplot_kw={'projection': 'polar'}, figsize=(7, 7))

# Independent variable for angle.
theta = np.arange(0, 2*np.pi, 0.01)

# This is a unit circle, since we know `sin(t)^2 + cos(t)^2 = 1`
r1 = np.sin(theta)**2 + np.cos(theta)**2

# A function that spirals outward.
r2 = np.sin(2*theta) - (2*theta)

# Plot the functions
ax.plot(theta, r1, lw=3, label="r = sin(t)^2 + cos(t)^2")
ax.plot(theta, r2, ls="--", lw=2, label="r = sin(2t)-2t")

# Remove all radial ticks and labels
ax.set_rticks([])

# Set the angle grids
ax.set_thetagrids([0, 90, 180, 270])

# Set the location to `lower right`.
plt.legend(loc="lower right")

plt.show()
```


90°
180°
0°
270°
r = $sin(\theta)^2 + cos(\theta)^2$
r = $sin(2\theta) - 2\theta$

12.3 Example of a contour plot and a heatmap

```
import numpy as np
import matplotlib.pyplot as plt

# Make a figure.
fig, ax = plt.subplots(figsize=(7, 7))

# Independent variables
r = 1.5
x = np.arange(-r, r, 0.01)
y = np.arange(-r, r, 0.01)

# Make a mesh grid, see the chapter on NumPy.
xx, yy = np.meshgrid(x, y)

# A multivariate (2D) Gaussian function
z = np.exp(-xx**2 - yy**2)

# Make a heatmap of `z` values
# `cmap` specifies the colormap
extent = [-r, r, -r, r]
plt.imshow(z, extent=extent, cmap="cividis")

# Draw the contours
plt.contour(xx, yy, z, colors="w", levels=5)

# Set the location of ticks for `x` and `y` axes
ticks = [-r, -0.5*r, 0, 0.5*r, r]
plt.xticks(ticks)
plt.yticks(ticks)

# Set the title and labels for axes
plt.title(r"z=exp(-x^2-y^2)")
plt.xlabel(r"x", fontsize=15)
plt.ylabel(r"y", fontsize=15)

plt.show()
```

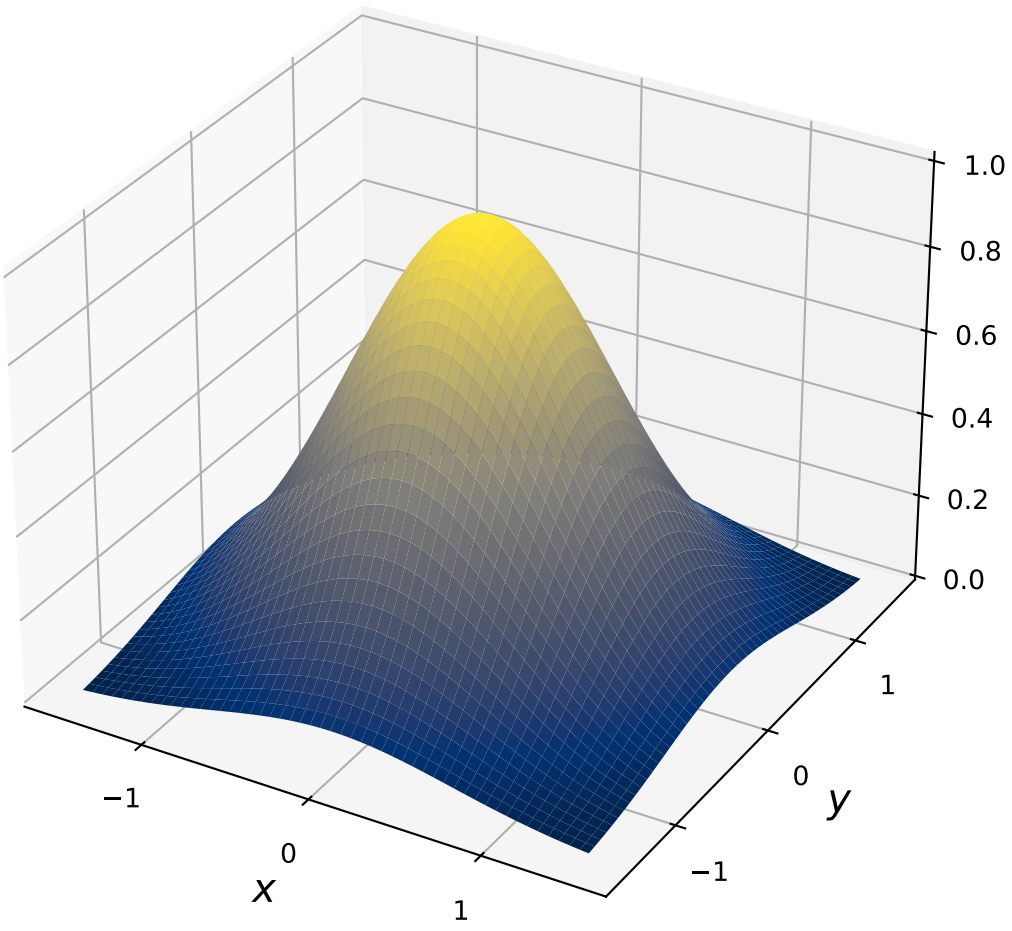

$z = \exp(-x^2 - y^2)$
1.50
0.75
0.00
−0.75
−1.50
y
−1.50
−0.75
0.00
0.75
1.50
x

12.4 Example of 3D plots

Let's see the same multivariate Gaussian function, but this time in 3D.

```
import numpy as np
import matplotlib.pyplot as plt

# We set the projection to `3d`.
fig, ax = plt.subplots(subplot_kw={'projection': '3d'}, figsize=(7, 7))

# Independent variables
r = 1.5
x = np.arange(-r, r, 0.01)
y = np.arange(-r, r, 0.01)

# Make a mesh grid, see the chapter on NumPy.
xx, yy = np.meshgrid(x, y)

# A multivariate (2D) Gaussian function
z = np.exp(-xx**2 - yy**2)

# Make a 3D surface of `z` values
# `cmap` specifies the colormap
ax.plot_surface(xx, yy, z, cmap="cividis")

# Set the location of ticks for `x` and `y` axes
ticks = [-1, 0, 1]
plt.xticks(ticks)
plt.yticks(ticks)

# Set the title and labels for axes
plt.title(r"z=exp(-x^2-y^2)")
plt.xlabel(r"x", fontsize=15)
plt.ylabel(r"y", fontsize=15)

plt.show()
```

$$z = \exp\left(-x^2 - y^2\right)$$



And the same function with its contours,

```
import numpy as np
import matplotlib.pyplot as plt

# We set the projection to `3d`.
fig, ax = plt.subplots(subplot_kw={'projection': '3d'}, figsize=(7, 7))

# Independent variables
r = 1.5
x = np.arange(-r, r, 0.1)
y = np.arange(-r, r, 0.1)

# Make a mesh grid, see the chapter on NumPy.
xx, yy = np.meshgrid(x, y)

# A multivariate (2D) Gaussian function
z = np.exp(-xx**2 - yy**2)

# Make a 3D wireframe for `z` values
ax.plot_wireframe(xx, yy, z, color="grey", lw=0.5)

# Add the 2D contour plots
# `cmap` specifies the colormap
ax.contourf(xx, yy, z, zdir='z', offset=0, cmap="cividis")
ax.contourf(xx, yy, z, zdir='x', offset=-r, cmap="cividis")
ax.contourf(xx, yy, z, zdir='y', offset=r, cmap="cividis")

# Set the location of ticks for `x` and `y` axes
ticks = [-1, 0, 1]
plt.xticks(ticks)
plt.yticks(ticks)

# Set the axes limits and labels
ax.set(xlim=(-r, r), ylim=(-r, r), zlim=(0, 1), xlabel='X', ylabel='Y', zlabel='Z')

# Set the title
plt.title(r"z=exp(-x^2-y^2)")

plt.show()
```

$$z = \exp\left(-x^2 - y^2\right)$$