Intermediate Python for Finance Professionals

Juan F. Imbet *Ph.D.*

Paris Dauphine University

About the course

- Final Exam: 100%

Week 1

Introduction to Python, grammar, syntaxis, history, binary arithmetic, good-practices, basic data-structures.

What is Python?

Python is a programming language that has been around for a while. However, only recently it has become popular for data science and machine learning applications.

Python is a:

1. **General purpose programming language**
2. **Interpreted language**
3. **Object-oriented language**
4. **High-level language**
5. **Dynamically typed language**

General purpose programming language

Python can be used for many different things. For example, you can use Python to build websites, create games, build machine learning models, analyze data, and more.

For example

1. **Web development** using Django and Flask
2. **Machine learning** using Scikit-Learn, TensorFlow, and Keras
3. **Data science** using Pandas, NumPy, and SciPy
4. **Game development** using Pygame
5. **Algorithmic trading** using Zipline and Quantopian
6. **AI applications** using OpenAI's API

Interpreted language

Python is an interpreted language, meaning that it is not directly translated into machine code. Instead, Python code is interpreted into Python bytecode, which is then interpreted (at runtime) into machine code.

Comparison to other programming languages

1. **C++** is a compiled language
2. **Java** is a compiled language
3. **Python** is an interpreted language
4. **JavaScript** is an interpreted language

Object-oriented language

- Python is an object-oriented language, which means that it provides features that support object-oriented programming (OOP). OOP is a programming paradigm that focuses on objects and data rather than actions and logic. For example, a car can be an object with properties like color, model, and mileage, and methods like drive and brake.

Comparison to other programming paradigms

1. **Procedural programming** is a programming paradigm that focuses on actions and logic rather than objects and data. For example languages like C and Fortran are procedural languages.
2. **Functional programming** is a programming paradigm that focuses on functions rather than objects and data. For example, languages like Haskell and Scala are functional languages.

High-level language

Comparison to other programming languages

1. **Low-level languages** are closer to machine languages than human languages. For example, assembly language is a low-level language.
2. **High-level languages** are closer to human languages than machine languages. For example, C++ is a high-level languages.

Dynamic vs Static Typing

Python is a dynamically typed language, meaning that variables do not have to be declared before they are used. The type of a variable determines how much memory is allocated for it, and what operations can be performed on it.

Comparison to other programming languages

1. **Dynamically typed languages** like Python and JavaScript do not require variables to be declared before they are used.
2. **Statically typed languages** like C++ and Java require variables to be declared before they are used.

Dynamic vs Static Typing

Static typing example, e.g. C++

```
int x = 1;
int y = 2;
double z = x / y;
char c = 'a';
```

Dynamic typing example, e.g. Python

```
x = 1
y = 2
z = x / y
c = 'a'
```

Python syntax

Python syntax is very clean, simple, and easy to understand. This is one of the reasons why Python is so popular for beginners.

Some unique features of Python syntax

1. **Indentation** is used to delimit blocks rather than curly braces
2. **No semicolons** are required to end statements
3. **No variable declarations** are required

Example of the Python syntax

```python
# This is a comment
x = 1
y = 2
if x < y:
    print('x is less than y')
else:
    print('x is greater than or equal to y')
```

SETTING UP YOUR ENVIRONMENT

Installing Python

There are two main versions of Python: Python 2 and Python 3. Python 2 is legacy, Python 3 is the current version. This course will use Python 3.

Installing Python

1. **Windows** - Anaconda
2. **Mac** - Anaconda
3. **Linux** - Anaconda
4. **Online** - Google Colab

Installing an IDE

An IDE (Integrated Development Environment) is a program that provides an editor, debugger, and compiler all in one. There are many different IDEs for Python, and it is important to choose one that you like. This course will use **VS Code**, but feel free to explore other options.

Installing VS Code

1. **Windows** - VS Code
2. **Mac** - VS Code
3. **Linux** - VS Code

Some useful extensions on VS Code

- Jupyter
- Python

Preliminaries

Computer Architecture and Programming Languages

How do Computers Work?

A computer is a machine that can be programmed to accept data (input), process it into useful information (output), and store it away (in a secondary storage device) for safekeeping or later reuse. The processing of input to output is directed by the software but performed by the hardware.

Computers think in binary

Computers are made of billions of tiny electronic components, which all work together to perform calculations. These components are called transistors. Transistors are made of semiconductors, which can be turned on and off by electricity. This is the basis of the binary system, which uses only two numbers (0 and 1) to represent all other numbers.

Binary numbers

Binary numbers are base 2 numbers, and have only two values – 0 and 1. The binary number system is a positional notation with a radix of 2. Each digit is referred to as a bit. The binary number system is also a positional notation numbering system, meaning that the position of the bit determines its value.

Formula for converting binary to decimal

$$\text{Decimal} = \sum_{i=0}^{n-1} d_i \times 2^i$$

where d_i is the ith digit of the binary number and n is the number of digits in the binary number. E.g. $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$

Binary code

Binary code is the language that computers use. It is made up of binary numbers (0s and 1s) that represent commands or other types of data. Different types of binary code can be used to represent text, images, audio, or other types of data.

Example: Substring of a long binary code adding two numbers
0101 1000 0000 0011 0000 0000 0000 0010

Binary code depends on the hardware of the computer. For example, the same binary code can represent different things on different computers.

Who has the time to write binary code?

Binary code is very difficult to write and understand. For this reason, programming languages were invented. The first programming language that **assembled** code into binary was **Assembly** in 1949.

Example of Assembly code:

```
MOV AX, 5
MOV BX, 10
ADD AX, BX
```

Who has the time to write Assembly code?

Assembly code is still very difficult to write and understand. For this reason, programming languages were invented. The first programming language that **compiled** code into binary was **FORTRAN** in 1957.

Example of FORTRAN code:

```
PROGRAM ADD
  INTEGER A, B, C
  A = 5
  B = 10
  C = A + B
END PROGRAM ADD
```

How is Python ran?

Python is an **interpreted** language, meaning that it is not directly translated into machine code. Instead, Python code is interpreted into Python bytecode, which is then interpreted (at runtime) into machine code.

Python bytecode is stored in files with the extension `.pyc`. These files are created by the Python interpreter when a `.py` file is imported. The bytecode is then executed by the Python virtual machine (PVM).

The PVM is a program that reads Python bytecode and executes it on the hardware. The PVM is written in C, which is a compiled language.

In what language is Python written?

Python is written in C, and some libraries are written in Python itself. This is why some C libraries can be used in Python.

C code

```c
#include <stdio.h>
int main() {
   printf("Hello, World!");
   return 0;
}
```

Python code

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

Running Python code

In VS Code using the Python extension running cells. Install the python and jupyter extensions.

In VS Code

```
Run Cell | Run Below | Debug Cell
#%%
print("Hello, World!")
```

In the terminal using the command `python <filename>`.

```
# hello.py
print("Hello, World!")
```

```
python hello.py
```

```
Hello, World!
```

Running Python code

Large code is normally written in a text editor and then executed in the terminal. This is the preferred way to write Python code. A good practice (and a necessary one to run code in e.g. parallel) is to check that the file is being run as the main program and not as a module.

```
# hello.py
if __name__ == '__main__':
    print("Hello, World!")
```

```
python hello.py
```

```
Hello, World!
```

Pseudo-code

Pseudo-code is a detailed description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language. Pseudo-code is used to plan out the structure of a program before writing actual code.

Some good practices that we will follow

Type hints

Type hints are a formalized way of adding type annotations to Python code. Type hints are not enforced by the Python interpreter, but they can be used by external tools such as type checkers, IDEs, etc.

```python
def add(x: int, y: int) -> int:
    return x + y
```

Checking the size of an object

```python
import sys
x = 1
sys.getsizeof(x)
```

```
28
```

Checking the type of an object

```python
type(x)
```

```
int
```

Checking the memory address of an object

```
id(x)
```

```
140735674332528
```

Testing if an object is of a certain type

```
isinstance(x, int)
```

```
True
```

Checking the documentation of an object

```
help(x)
```

```
Help on int object:
```

Looking at the source code of an object

```
import inspect
import pandas
inspect.getsource(pandas)
```

```
from __future__ import annotations\n\n__docformat...
```

Checking the attributes of an object

```
class Car:
    def __init__(self, color, model, mileage):
        self.color = color
        self.model = model
        self.mileage = mileage

car = Car('blue', 'BMW', 10000)
dir(car)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 ...
 'color',
 'mileage',
 'model']
```

In interactive mode, the interpreter will print the result of the last expression

```
1 + 1
```

```
2
```

I will abuse notation and display the output of some operations without the `print` function

```
x = 1
y = 2
x
y
```

```
1
2
```

Errors

Errors are a part of programming. Errors are caused by mistakes in the code, and they stop the program from being executed. Python has many built-in error types, and even allows you to create your own custom errors.

```
x = 1
y = 0
x / y
```

```
ZeroDivisionError: division by zero
```

Raise an error

```
raise ValueError('This is a value error')
```

```
ValueError: This is a value error
```

Exceptions

Exceptions are raised when the program is syntactically correct but the code resulted in an error. This causes the program to stop execution. Exceptions and errors can be handled using try-except blocks.

```
x = 1
y = 0
try:
    x / y
except ZeroDivisionError:
    print('Cannot divide by zero')
```

```
Cannot divide by zero
```

Warnings

Warnings are raised when the program is syntactically correct but there may be a problem. This does not cause the program to stop execution. Warnings can be ignored, or they can be turned into errors using the warnings module. Always be careful when ignoring warnings, as they may indicate a problem with the code. For example, never ignore warnings about matrix determinants being close to zero.

```
import warnings
warnings.warn('This is a warning')
```

```
<ipython-input-1-1a2b3c4d5e6f>:2: UserWarning: This is a warning
  warnings.warn('This is a warning')
```

Supress warnings

```
import warnings
warnings.filterwarnings('ignore')
```

Navigate through error messages

```python
x = 1
y = 0
x / y
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In [18], line 3
      1 x = 1
      2 y = 0
----> 3 x / y

ZeroDivisionError: division by zero
```

Python will tell you the type of error, the line of code that caused the error, and a description of the error. This information can be used to fix the error. However, sometimes the error message is not very helpful, and identifying the cause of the error comes with experience.

Being informative but not too verbose

Verbose means using more words than necessary. In programming, verbose normally refers to code that is longer than necessary, or that is intensive in the messages it prints.

A good practice is to be verbose if the user requires it.

```python
# An inadequate code

print("About to do something")
x = 1
print("Did something")
y = 2
print("Did something else")
z = x + y
print("Im done")
```

Let's start coding!

Byte and other notations

One byte equals 8 bits. A bit is a single binary digit, either a 0 or a 1. A byte can represent 256 different values, which is enough to represent all the letters in the English alphabet (both upper and lower case), the numbers 0-9, and some special characters. Base 2 is not the only way to represent numbers. Base 8 (octal) and base 16 (hexadecimal) are also commonly used.

```
x = 0b1010 # binary
y = 0o12 # octal
z = 0xA # hexadecimal
```

```
10
10
10
```

How to represent an integer in binary?

Zero followed by a lowercase b and then the binary representation , e.g. `0b1010`

```
x = 0b1010
```

```
10
```

How to represent an integer in octal?

Zero followed by a lowercase o and then the octal representation , e.g. `0o12`

```
x = 0o12
```

```
10
```

How to represent an integer in hexadecimal?

Zero followed by a lowercase x and then the hexadecimal representation , e.g. `0xA`

```
x = 0xA
```

```
10
```

We will not enter into details on the arithmetic and representation of numbers in different bases. Nevertheless we can switch between bases using the following functions

```
bin(10) # '0b1010'
oct(10) # '0o12'
hex(10) # '0xa'
```

Variables and data types

Variables are used to store data in a program. Variables can be thought of as containers that hold information. Their value (in Python) can be changed as the program runs (Dynamic Typing).

Python comes with many built-in data types, and you can define your own data types as well. The most common data types are strings, integers, floats, and booleans. Expressed in python as

```
x = 1
y = 2.5
z = 'Hello World'
w = True
```

Strings

Strings are used to store text. They are immutable, meaning that their value cannot be changed after they are created. Strings can be created using single quotes, double quotes, or triple quotes.

```
x = 'Hello World'
y = "Hello Again"
z = '''Hello World'''
```

Strings occupy memory, and the amount of memory they occupy depends on their length. Triple quotes are used to create multi-line strings.

Most common string methods

```python
x = 'HELLO world'
x.upper() # 'HELLO WORLD'
x.lower() # 'hello world'
x.title() # 'Hello World'
x.capitalize() # 'Hello world'
x.swapcase() # 'hello WORLD'
x.replace('world', 'universe') # 'HELLO universe'
x.count('l') # 3
x.startswith('H') # True
x.endswith('d') # True
x.find('l') # 2
x.find('z') # -1
x.index('l') # 2
x.index('z') # ValueError
x.isalnum() # False
x.isalpha() # False
x.isdecimal() # False
x.isdigit() # False
x.islower() # False
x.isupper() # False
x.isspace() # False
x.istitle() # False
x.split() # ['HELLO', 'world']
```

More methods

```
x = 'HELLO'
y = 'world'
x + y # 'HELLOworld'
x * 3 # 'HELLOHELLOHELLO'
x[0] # 'H'
x[1] # 'E'
x[-1] # 'O'
x[1:3] # 'EL'
x[1:] # 'ELLO'
x[:3] # 'HEL'
x[::2] # 'HLO'
x[::-1] # 'OLLEH'

' '.join([x, y]) # 'HELLO world'

x.find(y) # -1
```

Formatting strings

```python
x = 'Hello'
y = 'World'

print('Hello World')
print('Hello', 'World')
print('Hello ' + 'World')
print('Hello %s' % 'World')
print('Hello {}'.format('World'))
print(f'Hello {y}')
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Byte strings

Byte strings are used to store binary data. Byte strings can be created using the `b` prefix.

```
x = b'Hello World'
```

```
b'Hello World'
```

The main difference is that byte strings are already encoded, whereas regular strings are not. This means that byte strings can only contain ASCII characters, whereas regular strings can contain any Unicode characters.

ASCII vs Unicode

ASCII is a character encoding standard that uses 7 bits to represent all uppercase and lowercase letters, numbers, punctuation, and other symbols. ASCII is limited to 128 characters, which is enough to represent all the characters in the English alphabet.

Unicode is a character encoding standard that uses 8, 16, or 32 bits to represent all uppercase and lowercase letters, numbers, punctuation, and other symbols. Unicode is not limited to 128 characters, and can represent over 1 million characters.

Example

```
x = 'Hello World' # ASCII
y = '你好世界' # Unicode
```

Encoding and decoding strings

Encoding is the process of converting a string into a byte string. Decoding is the process of converting a byte string into a string. The default encoding is UTF-8, which uses 8 bits to represent each character.

```
x = 'Hello World'
y = x.encode()
z = y.decode()
```

```
b'Hello World'
'Hello World'
```

Documentation of string methods

```python
help(str)
```

```
Help on class str in module builtins:
```

Integers

Integers are used to store whole numbers. Integers can be created using the `int()` function, or by just writing a number with no decimal point.

```
x = 1
y = int('2')
```

All integers occupy the same amount of memory, regardless of their value. In a 64-bit system, integers occupy 28 bytes of memory.

```
import sys
sys.getsizeof(1)
```

```
28
```

Most common integer methods

```python
x = 1
x.bit_length() # 1
x.to_bytes(2, byteorder='big') # b'\x00\x01'
x.to_bytes(2, byteorder='little') # b'\x01\x00'
x.from_bytes(b'\x00\x01', byteorder='big') # 1
x.from_bytes(b'\x01\x00', byteorder='little') # 1
```

Methods between integers

```
x = 1
y = 2
x + y # 3
x - y # -1
x * y # 2
x / y # 0.5 - float
x // y # 0
x % y # 1
x ** y # 1
```

Documentation of integer methods

```
help(int)
```

```
Help on class int in module builtins:
```

Booleans

Booleans are used to store truth values. They can be created using the `bool()` function, or by just writing `True` or `False`. They are a subtype of integers, and `True` is equal to `1` and `False` is equal to `0`. They occupy the same amount of memory as integers. In many languages booleans occupy 1 bit of memory, but in Python they occupy 28 bytes of memory.

```
True
bool('True')
```

```
True
True
```

Boolean logic

Boolean logic is a branch of mathematics that deals with true and false values. Boolean logic is used to make decisions in computer programs. Boolean logic is based on the work of the English mathematician George Boole.

```
x = True
y = False
x and y # False
x or y # True
not x # False
```

Any complex logical expression such as xor can be expressed as a combination of and, or, and not.

Order of operations

The order of operations is the order in which mathematical expressions are evaluated. The order of operations is important because it can change the result of an expression. The order of operations is as follows:

1. Parentheses
2. Exponents
3. Multiplication and division
4. Addition and subtraction

```
x = 1
y = 2
z = 3
x + y * z # 7
(x + y) * z # 9
```

Floats

Floats are used to store decimal numbers. They can be created using the `float()` function, or by just writing a number with a decimal point. Interestingly enough, floats are stored in binary, and they occupy less memory than integers.

```
x = 1.0
y = float('2.5')
z = 1
import sys
sys.getsizeof(x) # 24
sys.getsizeof(z) # 28
```

```
24
28
```

Why do floats occupy less memory than integers?

Integers are stored in binary, and they occupy 28 bytes of memory. Floats are also stored in binary, but they occupy less memory because they are stored in scientific notation. This means that the decimal point is not stored, and the exponent is stored separately.

```
x = 1.0
y = 1e0
z = 1e1
sys.getsizeof(x) # 24
sys.getsizeof(y) # 24
sys.getsizeof(z) # 24
```

```
24
24
24
```

Most common float methods

```
x = 1.0
x.is_integer() # True
x.hex() # '0x1.0000000000000p+0'
x.as_integer_ratio() # (1, 1)
x.is_integer() # True
x.hex() # '0x1.0000000000000p+0'
x.as_integer_ratio() # (1, 1)
```

Methods between floats

```
x = 1.0
y = 2.5
x + y # 3.5
x - y # -1.5
x * y # 2.5
x / y # 0.4
x // y # 0.0
x % y # 1.0
x ** y # 1.0
```

Lists

Lists are used to store multiple items in a single variable. They can be created using square brackets, or by using the `list()` function. Lists are mutable, meaning that their values can be changed after they are created.

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

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

Lists can contain any type of data, including other lists.

```
x = [1, 2.0, True, [4, 'c', 6]]
```

```
[1, 2.0, True, [4, 'c', 6]]
```

Most common list methods

Many methods are performed inplace, i.e. they modify the list and return `None`

```
x = [1, 2, 3]
x[0] # 1 # The first element of the list is at index 0
x.append(4) # [1, 2, 3, 4]
x.extend([5, 6]) # [1, 2, 3, 4, 5, 6]
x.insert(0, 0) # [0, 1, 2, 3, 4, 5, 6]
x.remove(0) # [1, 2, 3, 4, 5, 6]
x.pop() # [1, 2, 3, 4, 5]
x.pop(0) # [2, 3, 4, 5]
x.clear() # []
```

More list methods

```
x = [1, 2, 3]
y = [4, 5, 6]
x + y # [1, 2, 3, 4, 5, 6] - concatenation
x * 3 # [1, 2, 3, 1, 2, 3, 1, 2, 3] - repetition
x.index(1) # 0
x.count(1) # 1
```

Look how lists and strings are similar. This is because strings are just lists of characters (also called chars).

```
x = ['1', '2', '3']
y = '123'
x[0] # '1'
y[0] # '1'
x[0] + x[1] # '12'
y[0] + y[1] # '12'
len(x) # 3
```

Dictionaries

Dictionaries are used to store key-value pairs. They can be created using curly braces, or by using the `dict()` function. Dictionaries are mutable, meaning that their values can be changed after they are created.

```
x = {'a': 1, 'b': 2, 'c': 3}
```

```
{'a': 1, 'b': 2, 'c': 3}
```

Dictionaries can contain any type of data, including other dictionaries.

```
x = {'a': 1, 'b': 2, 'c': {'d': 4, 'e': 5}}
```

```
{'a': 1, 'b': 2, 'c': {'d': 4, 'e': 5}}
```

Dictionaries in python are closely related to JSON objects, which are used to store data in web applications.

Most common dictionary methods

```
x = {'a': 1, 'b': 2, 'c': 3}
x['a'] # 1
x['d'] # KeyError
x.get('a') # 1
x.get('d') # None
x.keys() # dict_keys(['a', 'b', 'c'])
x.values() # dict_values([1, 2, 3])
x.items() # dict_items([('a', 1), ('b', 2), ('c', 3)])
x.pop('a') # {'b': 2, 'c': 3}
```

More dictionary methods

```
x = {'a': 1, 'b': 2, 'c': 3}
x['d'] = 4 # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
x.update({'e': 5, 'f': 6}) # {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
x.popitem() # {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
x.clear() # {}
```

Hash tables are closely related to dictionaries. They provide a "fast" way to look up values using keys. Hash tables are used in many programming languages, including Python.

Tuples

Tuples are used to store multiple items in a single variable. They can be created using parentheses, or by using the `tuple()` function. Tuples are immutable, meaning that their values cannot be changed after they are created.

```python
x = (1, 2, 3)
```

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

Tuples can contain any type of data, including other tuples.

```python
x = (1, 2.0, True, (4, 'c', 6))
```

```
(1, 2.0, True, (4, 'c', 6))
```

Tuples are immutable

```
x = (1, 2, 3)
x[0] = 0 # TypeError
```

```
TypeError: 'tuple' object does not support item assignment
```

Most common tuple methods

```
x = (1, 2, 3)
x[0] # 1
x.count(1) # 1
x.index(1) # 0
```

More tuple methods

```
x = (1, 2, 3)
y = (4, 5, 6)
x + y # (1, 2, 3, 4, 5, 6) - concatenation
x * 3 # (1, 2, 3, 1, 2, 3, 1, 2, 3) - repetition
```

Sets

Sets are used to store multiple items in a single variable. They can be created using curly braces, or by using the `set()` function. Sets are mutable, meaning that their values can be changed after they are created. A set only contains unique values, meaning that duplicates are not allowed.

```
x = {1, 2, 3}
```

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

Sets can contain any type of data, including other sets.

```
x = {1, 2.0, True, (4, 'c', 6)}
```

```
{1, 2.0, True, (4, 'c', 6)}
```

My own type: Overview of OOP

Python defines new types using the `class` keyword. Classes are used to create objects, which are instances of a class. Objects can have attributes and methods. Attributes are variables that belong to an object, and methods are functions that belong to an object.

```python
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print(f'Hello, my name is {self.name} and I am {self.age} years old')

john = Human('John', 28)
john.say_hello()
```

```
Hello, my name is John and I am 28 years old
```

Indentation

Indentation is used to delimit blocks of code in Python. This is different from other programming languages, which use curly braces to delimit blocks of code. Indentation is important because it determines which statements are executed in a program. Indentation is also important because it makes code easier to read. Keyboards have a tab key, which is used to indent code. Most code editors will automatically indent code for you. A python indentation is 4 spaces (also called 1 tab).

Indentation is important in Python! Always use an IDE that automatically indents your code. If you don't, you will get errors that are difficult to debug.

Control flow

Control flow is the order in which statements are executed in a program. Control flow statements are used to control the order in which statements are executed. The most common control flow statements are `if` statements, `for` loops, and `while` loops.

If, elif, and else statements

If statements are used to make decisions in a program. They allow the program to execute different code depending on whether or not a condition is true. If statements can be used by themselves, or they can be combined with `elif` and `else` statements.

```
x = 1
if x > 0:
    print('x is positive')
elif x < 0:
    print('x is negative')
else:
    print('x is zero')
```

```
x is positive
```

If statements can be alone or combined with elif and else statements

```python
x = 1
if x > 0:
    print('x is positive')
```

```
x is positive
```

```python
x = -1
if x > 0:
    print('x is positive')
elif x < 0:
    print('x is negative')
```

```
x is negative
```

Elif and else statements are optional, and cannot be used without an if statement

```
x = 0
elif x < 0:
    print('x is negative')
else:
    print('x is zero')
```

```
SyntaxError: invalid syntax
```

If statements in one line

Sometimes it is useful to write if statements in one line. This is called a ternary operator. It is useful when you want to assign a value to a variable depending on a condition.

```python
x = 1
y = 'positive' if x > 0 else 'negative'
y
```

```
'positive'
```

`__eq__` and `==` methods

We say that two objects are equal if they have the same value. In Python, the `==` operator is used to check if two objects are equal. The `==` operator is implemented using the `__eq__` method. This method can be overridden to change the behavior of the `==` operator.

```
class Human:
    ...
    def __eq__(self, other):
        return self.age == other.age

john = Human('John', 28)
jane = Human('Jane', 28)
john == jane # True
2==2.0 # True
```

```
True
True
```

`__lt__` and `<`, `__gt__` and `>`, `__le__` and `<=`, `__ge__` and `>=` methods

Mathematical operators that extend the natural order of numbers to other objects. These methods can be overridden to change the behavior of the mathematical operators.

```python
class Human:
    ...
    def __lt__(self, other):
        return self.age < other.age

    def __gt__(self, other):
        return self.age > other.age

    def __le__(self, other):
        return self.age <= other.age

    def __ge__(self, other):
        return self.age >= other.age

Human('John', 28) < Human('Jane', 28) # False
```

Loops

Loops are used to repeat code in a program. They allow the program to execute the same code many times without having to write it over and over again. The most common loops are `for` loops and `while` loops.

For loops

For loops are used to iterate over a pre-determined sequence of values. They allow the program to execute the same code many times without having to write it over and over again. For loops can be used to iterate over any sequence, including lists, tuples, dictionaries, and strings.

```python
x = [1, 2, 3]
for i in x:
    print(i)
```

```
1
2
3
```

For loops, range, and enumerate

```
x = [1, 2, 3]
for i in range(len(x)):
    print(i)
```

```
0
1
2
```

```
for i, j in enumerate(x):
    print(i, j)
```

```
0 1
1 2
2 3
```

`zip`

Loop over two or more sequences at the same time

```python
x = [1, 2, 3]
y = ['a', 'b', 'c']
for i, j in zip(x, y):
    print(i, j)
```

```
1 a
2 b
3 c
```

Can we mix methods?

`zip` and `enumerate`

```
x = [1, 2, 3]
y = ['a', 'b', 'c']
for i, j in enumerate(zip(x, y)):
    print(i, j)
```

```
0 (1, 'a')
1 (2, 'b')
2 (3, 'c')
```

One line `for` loops

Useful to create lists

```
x = [i for i in range(3)]
x
```

```
[0, 1, 2]
```

While loops

While loops are used to repeat code until a condition is no longer true. They allow the program to execute the same code many times without having to write it over and over again. While loops can be used to iterate over any sequence, including lists, tuples, dictionaries, and strings.

```python
x = 1
while x < 3:
    print(x)
    x += 1
```

```
1
2
```

Break statements

Break statements are used to exit a loop. They allow the program to exit a loop early if a certain condition is met.

```python
x = list(range(1000))
for i in x:
    if i == 3:
        break
    print(i)
```

```
0
1
2
```

Continue statements

Continue statements are used to skip the rest of a loop. They allow the program to skip the rest of a loop if a certain condition is met.

```python
x = list(range(4))
for i in x:
    if i == 2:
        continue
    print(i)
```

```
0
1
3
```

Pass statements

Pass statements are used to do nothing. But they are useful to complete a block of code that is not yet implemented, or create empty structures.

```python
class Human:
    pass
```

They are useful for `try` and `except` blocks when nothing should be done in the `except` block.

```python
try:
    x = 1 / 0
except ZeroDivisionError:
    pass # Not recommended
```

Modules

Most interesting things in Python do not come built-in. They are provided by external libraries called modules. Modules are files that contain Python code. They can be imported into your program using the `import` keyword. Modules can contain functions, classes, and other things. Optionally you can import only some functions from a module using the `from` keyword. You can also import all functions from a module using the `*` operator, and rename a module using the `as` keyword.

```
import math
from math import sqrt
from math import *
import math as m
from math import sqrt as s

math.sqrt(4) # 2.0
sqrt(4) # 2.0
s(4) # 2.0
m.sqrt(4) # 2.0
pi # from math module
```

Creating your own module

```python
# my_module.py
def add(x, y):
    return x + y

add(1,2)
```

```python
import my_module
my_module.add(1,2)
```

```
3
3 # Look that the function is called twice
```

Importing your own module

Use the `__name__` variable to check if a module is being imported or run directly. This is useful when you want to run a module as a script, but also import it into another module.

```python
# my_module.py
def add(x, y):
    return x + y

if __name__ == '__main__':
    print(add(1,2))
```

```python
#main.py
import my_module
my_module.add(1,2)
```

```
python main.py
```

```
3
```

Functions

Functions are used to perform a specific task. They allow the program to execute the same code many times without having to write it over and over again. Functions can be created using the `def` keyword. Functions can have parameters, which are variables that are passed into the function. Functions can also have return values, which are values that are returned by the function.

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

add(1, 2) # 3
```

Functions are objects

Functions are objects, and they can be passed around like any other object. This means that functions can be passed as arguments to other functions, and they can be returned by other functions. Parenthesis are used to call a function.

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

```
<function __main__.add(x, y)>
3
```

One line functions, lambda expressions

Sometimes it is useful to write functions in one line. This is called a lambda expression. It is useful when you want to pass a function as an argument to another function.

```
add = lambda x, y: x + y
add(1, 2) # 3
```

One line functions can also be created without an input

```
x = lambda: 1
x() # 1
```

Function kwargs

A keyword argument is an argument that is passed by name. They are useful when you want to have default values for some arguments. Keyword arguments can be passed in any order, and they can be used with any number of arguments.

```python
def add_numbers(x,y, verbose = False):
    if verbose:
        print(f'Adding {x} and {y}')
    return x + y

add_numbers(1, 2) # 3
add_numbers(1, 2, verbose=True) # 3
```

```
3
Adding 1 and 2
3
```

args and kwargs

Functions can have any number of arguments. This is useful when you don't know how many arguments a function will need. The `*` operator is used to pass a variable number of arguments to a function. The `**` operator is used to pass a variable number of keyword arguments to a function.

```
def add(*args):
    return sum(args)

add(1, 2, 3) # 6

def add(**kwargs):
    return sum(kwargs.values())

add(x=1, y=2, z=3) # 6

def add(*args, **kwargs):
    return sum(args) + sum(kwargs.values())

add(1, 2, 3, x=1, y=2, z=3) # 12
```

Logging

Logging is used to record information about a program's execution. It is useful for debugging, and it can also be used to record information about a program's execution. Logging is done using the `logging` module. The `logging` module has many built-in functions, and it can also be customized to suit your needs.

```python
import logging
logging.basicConfig(level=logging.INFO)
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

```
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

An appropriate optional logging

Use the fact that `and` statements are evaluated from left to right and stop as soon as a `False` value is found.

```python
import logging
logging.basicConfig(level=logging.INFO)

def some_function(verbose = False):
    if verbose:
        logging.info('This is an info message')
        logging.warning('This is a warning message')
        logging.error('This is an error message')
        logging.critical('This is a critical message')

    # equivalently
    verbose and logging.info('This is an info message')
    verbose and logging.warning('This is a warning message')
    verbose and logging.error('This is an error message')
    verbose and logging.critical('This is a critical message')
```

Other logging methods - write to a file

```
# Write to a file
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

Assertion

Assertions are used to check if a condition is true. They are useful for debugging, and they can also be used to check if a program is working as expected. Assertions are done using the `assert` keyword. The `assert` keyword has two arguments: a condition, and an optional message. If the condition is true, then the program continues. If the condition is false, then the program stops and an error is raised.

```python
x = 1
assert x == 1
assert x == 0, 'x should be 0'
```

```
AssertionError: x should be 0
```

Tests

Tests are used to check if a program is working as expected. They are useful for debugging, and they can also be used to check if a program is working as expected. Tests are done using the `unittest` module. The `unittest` module has many built-in functions, and it can also be customized to suit your needs. We will learn later to design tests using the `unittest` module. For now let's see an example to make sure the implicit function `sum` works as expected.

```
import unittest
class TestSum(unittest.TestCase):
    def test_sum(self):
        self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

if __name__ == '__main__':
    unittest.main()
```

```

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Basic User Input

User input is used to get input from the user. It is useful for getting information from the user, and it can also be used to get information from the user. User input is done using the `input` function. The `input` function has one argument: a prompt. The prompt is displayed to the user, and the user can enter a value. The value is returned by the `input` function.

```
x = input('Enter a number: ')
x
```

```
Enter a number: 1
'1'
```

Small Project: Temperature Converter

```python
def celsius_to_fahrenheit(celsius):
    return celsius * 9 / 5 + 32

def fahrenheit_to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5 / 9
```

Small Project: Temperature Converter

```python
def main():
    while True:
        try:
            temperature = float(input('Enter a temperature: '))
            break
        except ValueError:
            print('Please enter a number')

    while True:
        try:
            unit = input('Enter a unit: ')
            if unit.lower() == 'c':
                print(f'{temperature}°C = {celsius_to_fahrenheit(temperature)}°F')
                break
            elif unit.lower() == 'f':
                print(f'{temperature}°F = {fahrenheit_to_celsius(temperature)}°C')
                break
            else:
                print('Please enter a valid unit')
        except ValueError:
            print('Please enter a valid unit')
```

Small Project: Temperature Converter

```
if __name__ == '__main__':
    main()
```

```
Enter a temperature: 100
Enter a unit: c
100.0°C = 212.0°F
```

File I/O

File I/O is used to read and write files. It is useful for storing data. File I/O is done using the `open` function. The `open` function has two arguments: a filename, and a mode. The mode is used to specify how the file should be opened. The mode can be `r` for reading, `w` for writing, `a` for appending, `r` for reading and writing, `b` for binary, and `+` for updating. The default mode is `r`.

```python
# Writing to a file
with open('file.txt', 'w') as f:
    f.write('Hello World')

# Reading from a file
with open('file.txt', 'r') as f:
    print(f.read())
```

File I/O - Don't s

When dealing with files always use the `with` statement. This ensures that the file is closed properly. If you don't use the `with` statement, then you have to close the file manually.

```python
# Do not do this
f = open('file.txt', 'w')
f.write('Hello World')
f.close()
```

Decorators

Decorators are used to modify the behavior of a function. They allow the program to modify the behavior of a function without changing the function itself. Decorators are done using the `@` symbol. The `@` symbol is used to specify a decorator. The decorator is a function that takes a function as an argument, and returns a function. The decorator can be used to modify the behavior of the function.

```
def decorator(func):
    def wrapper(*args, **kwargs):
        # Do something before
        func(*args, **kwargs)
        # Do something after
    return wrapper
```

Useful decorators

Timing a function

```python
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        func(*args, **kwargs)
        end = time.time()
        print(f'{func.__name__} took {end - start} seconds')
    return wrapper

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

add(1, 2)
```

```
add took 0.0 seconds
```

Useful decorators

Logging a function

```
import logging
logging.basicConfig(level=logging.INFO)

def logger(func):
    def wrapper(*args, **kwargs):
        logging.info(f'Running {func.__name__} with args {args} and kwargs {kwargs}')
        func(*args, **kwargs)
    return wrapper
```

Timeit

Timeit is used to time a function using the `timeit` module.

```
import timeit
timeit.timeit('1 + 1') # input has to be string
%timeit 1 + 1 # Computes the average and s.d. of different runs.
```

```
0.01200000000000001
8.52 ns ± 0.00658 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

Type hints

Type hints are used to specify the type of a variable. They allow the program to specify the type of a variable without changing the variable itself. Type hints are done using the `:` symbol. The `:` symbol is used to specify a type hint. Hints are not enforced by the interpreter, but they are useful for documentation and debugging.

```python
def add(x: int, y: int) -> int:
    z: int = x + y
    return z

add(1, 2)
add(1.0, 2) # No error
```

Docstrings

Docstrings are used to document a function using the `"""` string. Try always to document your functions. Docstrings are useful for documentation and debugging. Use the following format for your docstrings. Most IDE will automatically show the docstring when you hover over a function and even help you to write it.

```python
def add(x: int, y: int) -> int:
    """Add two numbers

    Args:
        x (int): First number
        y (int): Second number

    Returns:
        int: Sum of x and y
    """
    z: int = x + y
    return z
```

Docstrings can be accessed using the function `help`.

Machine Epsilon

Machine epsilon is the smallest number that can be added to 1.0 and still be different from 1.0. It is useful for determining the precision of a floating point number. Machine epsilon is done using the `sys` module. The `sys` module has many built-in functions, and it can also be customized to suit your needs.

```
import sys
sys.float_info.epsilon # 2.220446049250313e-16
1.0 + sys.float_info.epsilon == 1.0 # False
1.0 + sys.float_info.epsilon/2 == 1.0 # True
```