Python: An Introduction

Juan F. Imbet *Ph.D.*

Paris Dauphine University

Master (M1) in Finance

A little bit about me

- Researcher in Financial Economics:
  - Social Media and Finance.
  - Tax non-compliance and corporate finance.
- Python enthusiast (for work and fun).
  - Textual analysis and Machine Learning.
  - Large Language Models (LLMs).
- Office P606, email: juan.imbet@dauphine.psl.eu
- Office hours by appointment.

Prerequisites (brush up if necessary!)

- Discrete Mathematics (sets, functions, relations, logic, integers, rational numbers).
- Linear Algebra (vectors, matrices, determinants).
- Statistics (mean, median, mode, variance, standard deviation, correlation, regression).

Roadmap

- **Session 1**: Getting Started and Introduction to Programming.
- **Session 2**: Intermediate Concepts.
- **Session 3**: Linear Algebra
- **Session 4**: Data Frames
- **Session 5**: Object Oriented Programming.
- **Session 6**: Computational Complexity.

Session 1

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. Everything in Python is an **object**. Objects have **attributes** and **methods**. Attributes are variables that belong to an object, and methods are functions that belong to an object.

Comparison to other programming paradigms

1. **Procedural programming**: Focuses on logic and procedures. E.g. C and Fortran.
2. **Functional programming**: Focuses on functions and data. E.g. Haskell and Lisp.

High-level language

It looks like English!

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. This means that variables do not have a fixed type. The type of a variable is determined at runtime, based on the value that is assigned to 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.

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

```
# 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

Rather than installing Python directly, it is recommended to install a Python distribution. A Python distribution is a collection of Python packages that are pre-installed and pre-configured. This makes it easier to get started with 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. (But I will not provide support for other IDEs).

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 (for now)

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

A number expressed in a specific base can be expressed as N_b, where N is the number and b is the base.

Example

$$150_{10} \rightarrow \text{The number 150 that we all know.}$$
$$\rightarrow 1 \times 10^2 + 5 \times 10^1 + 0 \times 10^0$$

How do we get the same number with powers of 2?

$$150_{10} \rightarrow \text{The number 150 that we all know.}$$
$$\rightarrow 1 \times 10^2 + 5 \times 10^1 + 0 \times 10^0$$

$$10010110_2 = 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 128 + 16 + 4 + 2 = 150_2$$

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.

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
```

But, who has the time (and skills) to write Assembly code?

(Very few people)
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
```

Look how it uses more human-readable words.

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.

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

```
python hello.py
```

```
Hello, World!
```

Interactive Mode:

Python can be run in interactive mode, which allows you to enter Python code directly into the Python interpreter. This is useful for testing small pieces of code, or for learning Python syntax.

```
1 + 1
```

```
2
```

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

```
x = 1
x
```

```
1
```

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

```
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.

Some good practices that we will follow

Type hints (more on types later)

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.

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

Checking the size of an object

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

```
28
```

Checking the type of an object

```
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:
```

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
```

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.

String Methods

We can modify strings by declaring them as variables and using the . notation

```
x = 'HELLO world'
x.upper() # 'HELLO WORLD'
x.lower() # 'hello world'
x.title() # 'Hello World'
```

More methods

```
x = 'HELLO'
y = 'world'
x + y # 'HELLOworld'
x * 3 # 'HELLOHELLOHELLO'
x[0] # 'H'
x[1] # 'E'
```

Formatting strings

```
x = 'Hello'
y = 'World'
print(f'{x} {y}')
```

```
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

```
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
```

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(10)
```

```
28
```

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.

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 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. Interestingly floats occupy less memory than integers. Floats can be created using the `float()` function, or by writing a number with a decimal point.

```
x = 1.0
z = 1
import sys
sys.getsizeof(x) # 24
sys.getsizeof(z) # 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
```

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.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.

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

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

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

```
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
```

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)}
```

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

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

```
x is positive
```

```
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.

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

```
'positive'
```

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.

```
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

```
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.

```
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.

```
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.

```
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.

```
def my_function():
    pass
```

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

```
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
```

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.

```
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.

```
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.

```
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.

```
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.

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

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

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'
```

Homework (Optional but HIGHLY Recommended)

- Don't waste your time using ChatGPT, you won't have access to it during the final exam.

Password Cracker

- A security system has a password that is 4 characters long. This password contains only ASCII characters (128 characters). The real password is unknown, but we know the *hash* of the password. The hash is a number that is the result of applying a function to the password, but it is impossible (very hard) to reverse the function.
- The hash of the password is `2868533088`, this integer is the result of applying the `hash_password()` function below to the password.
- Write a program that using brute force (trying all combinations) finds the password.
- For convenience the list `ascii_chars` contains all the ASCII characters that can be used in the password.
- Once you find the password, print it and break the loop. There are 128^4 = 268435456 possible combinations.

```
ascii_chars = [chr(i) for i in range(32, 127)]

def hash_password(text):
    hash=0
    for char in text:
        hash =  ( hash*281  ^ ord(char)*997) & 0xFFFFFFFF
    return hash
```