Python for Finance: An Introduction to Python

Amedeo Andriollo

Finance Department (DRM)
Université Paris Dauphine - PSL

Structure of the course

- Email: amedeo.andriollo@dauphine.psl.eu
- Grading: 100% final exam.
- Office hours: feel "free" to (DO) come: B612.
- Slides/materials are updated regularly. Report a typo/mistake: corrections help the class. Slides are partially based on last year's (prof. Imbet).
- This session:
  - Intro;
  - Setting up your environment;
  - Preliminaries;
  - Control Flow;
  -

▷ Next session:

What is Python?

- Python is a programming language that has been around since the 80s.
- Its name derives from "Monty Python's Flying Circus".

↪ very popular for data science and machine learning applications (and even more with AI).

- 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:

  ...from building websites, to create games..

  ...from tuning machine learning models to visualize data...

  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.

↪ Python code is interpreted into *Python bytecode*, then interpreted (at runtime) into machine code.

Comparison to other programming languages

1. C++, Java are compiled languages:

   A (separate) program, the *compiler*, reads the source code, all at once, and creates an executable. [Faster]

2. Python, JavaScript are interpreted languages:

   A program called an *interpreter* reads and executes the raw source code line-by-line during runtime. [Slower]

Object-oriented language

- Python is an object-oriented language: everything in Python is essentially an object, an instance of a class, containing data and performing actions.
  - Class = croissant recipe VS. Object = croissant made using that recipe.

  ↪ Multiple croissants with the same recipe, and different input to make different viennoiserie.
- **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.
- Example: I want to drive it at max power for 2" and turn the steering leftwards by 15 degrees.
  - Procedurally:
    1. Set drive motor voltage to 5 volts;
    2. Sleep for 2";
    3. Set the steering servo to whatever produces a leftwards change;
    4. Wait until my steering servo reports 15 degrees left.
  - Object-orientally:
    1. The CarController object makes a call to the Motor object, requesting max power;
    2. The CarController sleeps for 2";
    3. The CarController makes a call to the Steering object, requesting it...

High-level language

- It looks like English!
- **Comparison to other programming languages**:
  - Low-level languages are closer to machine languages than human languages (e.g., Assembly language).
  - High-level languages are closer to human languages than machine languages (e.g., C++, yet debatable).
- Not a binary choice: there is a spectrum between closer to the hardware (low) and further from the hardware (high).

Dynamic vs Static Typing

- Python is a dynamically typed language.
  $\hookrightarrow$ 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 do not require variables to be declared before they are used.
  2. Statically typed languages require variables to be declared before they are used (e.g., C++).

Python syntax

- Python syntax is very clean, simple, and easy to understand: intuitive.
- **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 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: will work with Python 3 is the current version (3.1X.y).
- 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 preconfigured (easier to get started).
  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 manyIDEs for Python, feel free to explore: choose one that you like.
- This course will use VS Code: support for other IDEs will not provide.
- **Installing VS Code**
  1. Windows - VS Code
  2. Mac - VS Code
  3. Linux - VS Code
- Extensions on VS code: Jupyter, Python.

Preliminaries

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 i) directed by the software but ii) performed by the hardware.
- Computers are made of billions of tiny electronic components, which all work together to perform calculations (transistors):
  $\hookrightarrow$ those are turned on and off by electricity: the basis of the binary system.

Binary numbers

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

$$150_{10} = 1 \times 10^2 + 5 \times 10^1 + 0 \times 10^0.$$

  $\hookrightarrow$ The number 150 is the one we all know.
- How do we get the same number with powers of 2?

$$\begin{aligned} 10010110_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 128 + 16 + 4 + 2 = 150_{10}. \end{aligned}$$

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
```

Who has the time(/skills) to code in Assembly?

- Assembly code is very difficult to write/understand: 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: first interpreted into Python bytecode and then 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!
```

- Large code is normally written in a text editor and then executed in the terminal.
  This is the preferred way to write Python code.

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: you are not (yet) a machine!
- 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 vs. can be turned into errors using the warnings module.

! Always be careful when ignoring warnings, as they may indicate a problem with the code (e.g., matrix determinants ≈ 0).

```
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: info that can be used to fix the error.
- Sometimes might not be very helpful, then experience+patience.

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 property of an object

- 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
```

Checking the property of an object

- Testing if an object is of a certain type:

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

```
True
```

- 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 Py) can be changed as the program runs (i.e., 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 integers, floats, strings and booleans.

```
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: its amount 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'
```

```
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.

↪ Byte strings can only contain ASCII characters
VS. 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 (other alphabets as well).

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 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.
- Same amount of memory as integers: in Python, 28 bytes (VS. 1 byte).
- Boolean logic is a branch of mathematics that deals with true and false values. Boolean logic is used to make decisions in computer programs.

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

- Any complex logical expression can be expressed combining: **and**, **or**, and **not**.

Order of operations

- The order of operations is the order in which mathematical expressions are evaluated.
  ↪ The order 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, created using the `float()` function, or simply by writing a number with a decimal point.

```
x = 1.0
z = 1
import sys
sys.getsizeof(x) # 24
sys.getsizeof(z) # 28
```

- Interestingly floats occupy less memory than integers:
- Integers are stored in binary and Floats as well
  Floats are stored in scientific notation, occupying less memory.
  ↪ 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
```

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
```

- Lists and strings are similar: 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}
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.

Tuples

- Tuples are used to store multiple items in a *single* variable.
  Created using parentheses, or by using the `tuple()` function.

```
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))
```

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

- Tuples are immutable: their values cannot be changed after they are created.

```
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)}
```

! The difference between Set VS. Tuple VS. List:
  - Set: mutable, unordered, uniqueness (duplicates are removed).
  - Tuple: immutable, unordered, allow duplicates.
  - List: mutable, ordered, allow duplicates.

Indentation

- Indentation is used to delimit blocks of code in Python, different from other programming languages, which use curly braces to delimit blocks of code.
- Indentation is important in Python!
  - it determines which statements are executed in a program; plus, readability;
  - keyboards have a tab key, which is used to indent code: a python indentation is 4 spaces (also called 1 tab).

! Always use an IDE that automatically indents your code.
(Most code editors will automatically indent code)
$\hookrightarrow$ otherwise you will get errors difficult to debug.

Control Flow

If, elif, and else statements

- "If" statements are used to make decisions in a 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 can be alone or combined with elif and else

```
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 but cannot be used without an if

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

```
SyntaxError: invalid syntax
```

- 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'
```

For 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 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
```

- One line for loops: useful to create lists

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

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

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')
```

While loops

- While loops are used to repeat code until a condition is no longer true, allowing the program to execute the same code many times.
- 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/Continue statements

- Break statements are used 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 are used 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.

↪ 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,
  ↪ provided by external libraries called modules.
- Modules are files that contain Python code (e.g., functions, classes,).
- They can be imported into your program using the `import` keyword.
  - You can import only some functions from a module using the `from` keyword.
  - You canimport 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, 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:
  - Inputs: parameters/variables that are passed into the function.
  - Outputs: return values that are returned by the function.

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

add(1, 2) # 3
```

Functions as objects and lambda expressions

- 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
```

- 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 (KeyWord Arguments)

- A keyword argument is an argument that is passed by name.
  ↪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
```

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

args and kwargs

- Functions can have any number of arguments. If 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 (useful for debugging).
- Logging is done using the logging module (built-in functions customizable).

```
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 facts that: i) `and` statements are evaluated from left to right and ii) 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.
  Useful for debugging, and used to check if a program is working as expected.
- Assertions are done using the `assert` keyword with respect to two arguments: i) a condition, and ii) an optional message.
  ↪ If the condition is true (false), the program continues (stops with an error).

```
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: useful for getting info from the user.
- User input is done using the `input` function that 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'
```

File I/O

- Useful for storing data, File I/O is used to read and write files, called by the `open` function.
- The `open` function has two arguments: i) a filename, and ii) a mode, used to specify how open the file. ↪ The mode can be: `r` for reading (default), `w` for writing, `a` for appending, `b` for binary, and `+` for updating.

```
# 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())
```

- 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.

```
# 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 without changing the function itself.
- Decorators are done using the `@` symbol.
  ↪ the decorator is a function that takes a function as an argument, and returns a function.

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

Timing a function

```
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
```

- `timeit` is used to time a function.

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

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
```

Type hints

- Type hints are used to specify the type of a variable, without changing it.
- Type hints are done using the : symbol. Hints are not enforced by the interpreter, butuseful for documentation and debugging.

```
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.

```
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.

```
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
```

Homework (Optional but HIGHLY Recommended)

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

! Use it to learn.

Password Cracker

- A security system has a password that is 4 characters long, with only ASCII characters (128 characters).
- The real password is unknown, but we know the hash of the password: the result of applying a function to the password.

! 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 $128^4 = 268'435'456$ possible combinations.

Password cracker: helper function

```
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
```