Python: Object-Oriented Programming

Hugo Mougard

https://www.orsys.fr/

Table of Contents

About the Seminar

Introduction

Name	Hugo Mougard
Email	hugo@mougard.fr
Activity	Senior Machine Learning Engineer. Python, ML & MLOps Freelance
Specialization	Natural Language Processing & Machine Learning applied to source code

Requirements

- Basic knowledge of programming
- Have a **Google** account in order to acces the labs environments in Google Colaboratory

Objectives

- Master the syntax of the Python language
- Acquire the essential notions of object-oriented programming
- Know and implement different Python modules
- Implementing tools for testing and evaluating the quality of a Python program

Resources

I will give you links to the resources used before and/or after each day.

They are also accessible through My Orsys.

Schedule

4 days, 9AM to 12:30PM and 1:30PM to 5PM.

Introduce yourself!

- Your name
- Your job
- Your client company if applicable
- The skills and knowledge you have related to this training session
- Your objectives and expectations for this training session

Python Language

History

1989 Creation of the language by Guido Van Russum

2001 Launch of the Python Software Foundation

2001 Transition to GPL

2009 Python 3

Logo Python, Python Software Foundation (”PSF”), PSF trademark.

Characteristics

Python is:

Interpreted and compiled on-the-fly, with C modules

Object-oriented (but not limited to)

Portable Compatible with all current platforms

Flexible Swiss Army knife, from system administration to web development

Popular Top 5 most used languages for years

Strengths/Weaknesses

Strengths

- Stable
- Cross-platform
- Easy to learn
- Large community (most used since 2019)
- Need something? There's a module for it.

Weaknesses

- Not compiled
  - Slower than low-level languages
  - Optimizing an operation ⇒ not easy to learn

Platforms

Different interpreters:

- CPython/Pypy $\Rightarrow$ C/C++
- Jython $\Rightarrow$ JVM
- IronPython $\Rightarrow$ .Net

Domains

Application domains:

- Web (Django, Flask, ...)
- Sciences (Data mining, Machine learning, Physics, ...)
- OS (Linux, Raspberry Pi, System administration scripting, ...)
- Education (Introduction to programming)
- 3D CAD (FreeCAD, pythonCAD, ...)
- Multimedia (Kodi, ...)

Development Environments

The three main Python IDEs are:

Visual Studio Code Free IDE offered by Microsoft

PyCharm IDE with both free and professional versions, offered by JetBrains

Spyder Open-source and free IDE focused on data science

They all integrate modern development tools.

However, you can of course use your favorite text editor (emacs, vim, atom, sublime text, etc.).

Python Versions 2 and 3

Version 2 No longer supported since January 1, 2020. Still present in legacy systems.

Version 3 Current version of Python
. All new developments should use it.

Resources

- Official Python Documentation
- Python Learning Reddit (forum)
- Course support, download the PDF
- Python Tutor

Installation

Refer to the excellent blog post on Real Python about the topic.

Do you have any questions ?

Python Fundamentals

Python Fundamentals

Basic Types and Variables

Introduction to variables

A variable is a *label* assigned to a Python object.

We can use this label instead of the object it refers to at any time in our programs.

Variable creation

To create a variable, we use the following syntax:

```
answer: int = 42
```

Variable usage

To use a variable, we just refer to it by its name:

```
>>> answer: int = 42
>>> print(answer)
42
```

Variable naming

There are two rules to respect when it comes to variables names:

- The names should start by a lowercase letter, uppercase letter or an underscore
- The names should continue by those same characters and you can also mix in digits

Unicode

From Python 3 on, it's possible to use any unicode letter in variable names. *It's strongly discouraged. Do yourself a favor and stick to ASCII!*

Basic types

To represent simple values, we mainly use:

`int`	Integers
`float`	Reals
`str`	Strings
`bool`	Booleans

The `type` function

Useful to know the type of an object:

```
>>> question = "What's the answer to the universe?"
>>> answer = 42
>>> type(question)
<class 'str'>
>>> type(answer)
<class 'int'>
```

Type conversion

We can use the classes that represent basic values to convert *to* them:

```
>>> float(42)
42.0
```

Do you have any questions ?

Python Fundamentals

Basic Types and Variables

Labs

Labs

Instructions

Python Fundamentals

Display

Display function

`print` is the main function to display text in Python:

```
>>> print("Hello World!")
Hello World!
```

`print` arguments

`sep` Displayed between each argument. Default: a space

`end` Displayed after the last argument. Default: a newline

`file` Output file. Default: `sys.stdout`

`flush` Should we flush the buffer? Default: `False`

```
>>> print("Hello", "World", sep=", ", end="!\n")
Hello, World!
```

Formated strings

It's possible to build complex strings with the `format` method:

```
>>> "My name is {lastname}, {firstname} {lastname}.".format(
...     firstname="James",
...     lastname="Bond")
'My name is Bond, James Bond.'
```

The website PyFormat gives more examples on available options.

f-strings

`format` has a handy shortcut:

```
>>> firstname = "James"
>>> lastname = "Bond"
>>> f"My name is {lastname}, {firstname} {lastname}."
'My name is Bond, James Bond.'
```

Do you have any questions ?

Python Fundamentals

Display

Labs

Labs

Instructions

Python Fundamentals

Collections

Introduction

The 4 main data structures in Python are:

`list`	Mutable lists, can be heterogeneous
`dict`	Key-value associations
`tuple`	Immutable tuples
`set`	Sets (no repetition, unordered)

`list` characteristics

- Probably the most used collection in Python
- Can contain elements of different types (*not recommended at all*)
- Good performance
- Despite its name, it's closer to an array than to a linked list (e.g. $\mathcal{O}(1)$ access)
- Not suited for scientific computations (see `numpy` instead)

`list` usage

```
>>> l = []                  # Create an empty list
>>> l = [1, 2, 3]           # Create a list with elements
>>> l[0]                    # Positive indices
1
>>> l[1]
2
>>> l[-1]                   # Negative indices
3
>>> l[-2]
2
>>> l.append(4)             # Add an element
>>> l.extend([5, 6])        # Add several elements
>>> l.sort(reverse=True)    # Sort, here reversed
>>> len(l)                  # Size
6
>>> del l[4]                # Deletion by index
>>> l.remove(1)             # Deletion by value
>>> l
[6, 5, 4, 3]
```

Slices

A list slice is a subset of a list:

```
>>> l: list[str] = ["a", "b", "c", "d", "e", "f"]
>>> l[0:3]
["a", "b", "c"]
>>> l[:3]
["a", "b", "c"]
>>> l[3:]
["d", "e", "f"]
>>> l[0:3:2]
["a", "c"]
```

`dict` characteristics

`dict` allows to define key-value associations:

- Equivalent to JS objects, Java maps
- Access the value associated to a key in $\mathcal{O}(1)$
- Test if a key is in the dict in $\mathcal{O}(1)$

`dict` usage

```
>>> d = {}                                # Empty dict creation
>>> d = {"a": 1, "b": 2}                  # Creation with elements
>>> d = dict([("a", 1), ("b", 2)])        # Creation from an iterable
>>> d = dict(a=1, b=2)                    # Creation with keyword style
>>> d["a"]                                # Accessing a value
1
>>> d["c"] = 3                            # Modifying or adding an entry
>>> "a" in d                              # Testing presence
True
>>> "d" in d
False
>>> 1 in d
False
>>> len(d)                                # Size
3
>>> del d["a"]                            # Deletion
>>> d
{"b": 2, "c": 3}
```

`tuple` characteristics

Very similar to a list, but:

- Immutable
- Suited for elements of different types
- Hashable if its elements are hashable
- Standard way to return several values from a function

`tuple` usage

```
>>> t = ()          # Empty tuple creation
>>> t = (1, 2, 3)  # Creation with elements
>>> t[0]           # Positive indices
1
>>> t[1]
2
>>> t[-1]          # Negative indices
3
>>> t[-2]
2
>>> 1 in t         # Presence test
True
>>> 4 in t
False
>>> t + (4, 5, 6)  # Concatenation
(1, 2, 3, 4, 5, 6)
>>> len(t)         # Size
3
```

Multiple definitions with a `tuple`

```
>>> a, b = 0, 1
>>> a
0
>>> b
1
```

Multiple values return with a `tuple`

```
>>> def sum_and_product(a, b):
...     return a + b, a * b
...
>>> x, y = sum_and_product(3, 4)
>>> print(f"Sum: {x}, product: {y}")
Sum: 7, product: 12
```

`set` characteristics

`set`s are unordered and without repetition and provide:

- Math set operations (union, intersection, ...)
- Fast presence test ($\mathcal{O}(1)$ vs $\mathcal{O}(n)$ for a `list`)

```
>>> import timeit
>>> timeit.timeit(setup="l = list(range(1_000_000))",
...               stmt="1_000_000 in l",
...               number=1_000)
6.942514133999794
>>> timeit.timeit(setup="s = set(range(1_000_000))",
...               stmt="1_000_000 in s",
...               number=1_000)
2.4280000616272446e-05
```

`set` usage

```
>>> s = set()                          # Empty set creation
>>> s = {"a"}                          # Creation with elements
>>> s = set(["a"])                     # Creation from an iterable
>>> s.add("b")                         # Add an element
>>> s.update(["c", "d"])               # Add several elements
>>> s
{'c', 'b', 'd', 'a'}
>>> "a" in s                           # Test presence
True
>>> "e" in s
False
>>> s | {"e", "f"}                     # Union
{'c', 'b', 'e', 'a', 'f', 'd'}
>>> s & {"a", "e"}                     # Intersection
{'a'}
>>> s - {"a", "e"}                     # Difference
{'d', 'c', 'b'}
>>> s < {"a", "b", "c", "d", "e"}      # Subset test
True
```

Immutable `sets`

`frozenset` is an immutable set alternative.

Similar to what `tuple` is compared to `list`.

Do you have any questions ?

Python Fundamentals

Collections

Labs

Labs

Instructions

Python Fundamentals

Control Structures

Introduction

Python has 3 main control structures: `if`, `while` and `for`.

We'll talk about `try`/`except` later to handle errors.

From Python 3.10 on, there's also a `match` structure but it has few good use cases, we will not study it here.

if

```
>>> age = 23
>>> if age < 18:
...     print("Minor in France")
... elif age < 21:
...     print("Minor in the US")
... else:
...     print("Adult")
...
Adult
```

`for` · With indices

The `range` function can be used to generate indices for iteration:

```
range(10)                # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(1, 10)             # 1, 2, 3, 4, 5, 6, 7, 8, 9
range(1, 10, 2)          # 1, 3, 5, 7, 9
list(range(1, 10, 2))  # [1, 3, 5, 7, 9]
```

Usage in a loop:

```
>>> for i in range(2):
...     print(i)
...
0
1
```

for · With a list

```
>>> l = ["a", "b"]

>>> for item in l:
...     print(item)
...
a
b

>>> for i in range(len(l)):
...     print(i, l[i])
...
0 a
1 b

>>> for i, item in enumerate(l):
...     print(i, item)
0 a
1 b
```

`for` · With a dict

```
>>> d = {"a": 1, "b": 2}

>>> for key in d:
...     print(key)
...
a
b

>>> for key, value in d.items():
...     print(key, value)
...
a 1
b 2
```

`for` · With several iterables

```
>>> l1 = ["a", "b", "c"]
>>> l2 = [True, False]

>>> for item1, item2 in zip(l1, l2):
...     print(item1, item2)
...
a True
b False

>>> import itertools
>>> for item1, item2 in itertools.zip_longest(l1, l2):
...     print(item1, item2)
...
a True
b False
c None
```

Works with more than 2 iterables.

while

```
>>> ok = False
>>> while not ok:
...     answer = input("Type y or n: ")
...     ok = answer in {"y", "n"}
...
Type y or n: yes
Type y or n: y
>>> answer
'y'
```

break & continue

break Used to skip to after the most inner loop

continue Used to skip to the next iteration of the most inner loop

```
>>> i = 0
>>> while True:
...     i += 1
...     if i < 10:
...         continue
...     if i > 11:
...         break
...     print(i)
...
10
11
```

`else` branch for loops

The `else` branch is executed if no `break` was used in the loop body:

```
>>> for i in range(1, 4):
...     print(i)
... else:
...     print("Soleil")
...
1
2
3
Soleil

>>> for i in range(1, 4):
...     if i == 3:
...         break
...     print(i)
... else:
...     print("Soleil")
...
1
2
```

Do you have any questions ?

Python Fundamentals

Control Structures

Labs

Labs

Instructions

Python Fundamentals

Reading and Writing Files

Main tools

- `open` function to retrieve an object to manipulate a file
- Usage of a `with` statement to ensure that file closing is handled properly

`with` statement

Introduces a context:

- Entering and leaving the context will execute some code, here the opening and closing of the file
- Errors can be handled

Most important open parameters

`mode` By default, `"r"` for reading. See here for all the options.

`encoding` By default, locale dependent. Nowadays, `encoding="utf8"` is almost always what you want.

Reading

```
>>> with open("example.txt", encoding="utf8") as fh:
...     print(fh.read())
...
My first line
My second line

>>> with open("example.txt", encoding="utf8") as fh:
...     for line in fh:
...         print(line)
...
My first line

My second line


>>> with open("example.txt", encoding="utf8") as fh:
...     lines = fh.readlines()
...     print(len(lines))
...
2
```

Writing

```
with open("example.txt", mode="w", encoding="utf8") as fh:
    fh.write("My text\n")

with open("example.txt", mode="w", encoding="utf8") as fh:
    fh.writelines(["My first line\n", "My second line\n"])
```

Manipulating two files at once

```
with open("example.txt", encoding="utf8") as fh_read, open(
    "example-upper.txt", mode="w", encoding="utf8"
) as fh_write:
    fh_write.write(fh_read.read().upper())
```

Newlines

By default, Python 3 understands newlines from all platforms.

When writing, it will use mostly `\n` as a default option.

Do you have any questions ?

Python Fundamentals

Reading and Writing Files

Labs

Labs

Instructions

Python Fundamentals

Functions

Introduction

Functions are one of the two main ways (with classes) to organize code.

Example

```
>>> def add(a: int, b: int) -> int:
...     return a + b
...
>>> add(1, 2)
3
```

Default values

```
>>> def add(a: int, b: int = 1) -> int:
...     return a + b
...
>>> add(1, 2)
3
>>> add(1)
2
```

Mutable default values

Warning

Never give a mutable default value to a function.

```
>>> def append_to(item: int, lst: list[int] = []) -> list[int]:
...     lst.append(item)
...     return lst
...
>>> append_to(1)
[1]
>>> append_to(1)
[1, 1]
```

Keyword style

```
>>> def sub(a: int, b: int) -> int:
...     return a - b
...
>>> sub(3, 2)
1
>>> sub(a=3, b=2)
1
>>> sub(b=3, a=2)
-1
```

Variables scope

In Python, the variables scope is the function (not the block).

```
>>> s: int = 3
>>> def square(n: int) -> int:
...     s = n ** 2
...     return s
...
>>> s
3
>>> square(3)
9
>>> s
3
```

Do you have any questions ?

Python Fundamentals

Functions

Labs

Labs

Instructions

Python Fundamentals

Object Oriented Programming

Introduction

Object Oriented Programming is a style of programming with the following goals:

- encapsulation
- modularity
- reusability

Key idea

Define objects that group together:

- data
- methods related to the data

Objects definition

To define objects, we write a class — a “recipe” to create an object (we also call objects instances) :

```python
from math import sqrt


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

    def distance(self, other):
        return sqrt((other.x - self.x) ** 2 + (other.y - self.y) ** 2)
```

- `Point` is the name of the class
- `__init__` is a special method to initialize the object
- `self.x` & `self.y` are instance attributes
- `distance` is an arbitrary method we’ve defined

Objects usage

Once the class is defined, we can use it as any other class in Python (e.g. `list`):

```
>>> from math import sqrt
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def distance(self, other):
...         return sqrt((other.x - self.x) ** 2
...                     + (other.y - self.y) ** 2)
...
>>> point1 = Point(2, 2)
>>> point2 = Point(-2, -2)
>>> point1.distance(point2)
5.656854249492381
```

Difference between instance attributes and class attributes

```
>>> class Point:
...     coeff = 5  # Class attribute, accessible to all instances
...     def __init__(self, x, y):
...         self.x = x  # Instance attribute
...         self.y = y  # Instance attribute
...
>>> point1 = Point(1, 2)
>>> point2 = Point(3, 4)
>>> point1.x, point1.y, point1.coeff
(1, 2, 5)
>>> point2.x, point2.y, point2.coeff
(3, 4, 5)
```

If an instance attribute has the same name as a class attribute, it will shadow it.

A few important remarks

- The first argument is named `self` by convention
- No notion of visibility in Python (private attributes or methods)
- Instead, we say that an attribute or a method is private by using a convention: adding a `_` in front of the name (`self._x`)

Polymorphism

Adaptation of the behaviour of functions or operators depending on the objects they are applied to.

E.g., the `+` operator in Python can mean:

- String concatenation
- List concatenation
- Integer addition
- ...

⇒ A different method is called in each case!

Warning

We also use the term polymorphism for simple functions or class methods, not only for operators.

Example of polymorphism by operator overloading

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __add__(self, other):
...         return Point(self.x + other.x, self.y + other.y)
...
>>> point1 = Point(2, 3)
>>> point2 = Point(3, 4)
>>> point3 = point1 + point2
>>> point3.x, point3.y
(5, 7)
```

Inheritance

To define a class, we can start from an existing class instead of from scratch. It's called inheriting from a class:

```
>>> class Rectangle:
...     def __init__(self, width, height):
...         self.width = width
...         self.height = height
...     def area(self):
...         return self.width * self.height
...
>>> class Square(Rectangle):
...     def __init__(self, side):
...         super().__init__(side, side)
...
>>> Square(2).area()
4
```

Multiple inheritance

It's possible to inherit from several classes, not only one.

- The order in which the parent classes are queried to find a function with `super()` is called MRO (*Method Resolution Order*)
- It's possible to access the MRO of a class by its `__mro__` attribute

Warning

Using multiple inheritance is common but can lead to very complex code.

Do you have any questions ?

Python Fundamentals

Object Oriented Programming

Labs

Labs

Instructions

Python Fundamentals

Error Handling

Introduction

Python handles errors with a special construct: `try`/`except`, and many different exception classes.

Native exception hierarchy

The Python doc contains a full list.

Intercept an error

```
>>> def div(a, b):
...     try:
...         return a / b
...     except ArithmeticError:
...         return float("nan")
...
>>> div(4, 3)
1.3333333333333333
>>> div(4, 0)
nan
```

Raise an error

```
>>> wind_speed = -9999
>>> if wind_speed < 0:
...     raise ValueError("Speed cannot be negative")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Speed cannot be negative
```

Define a custom error

It's possible to inherit from `Exception` or a more specific exception class.

```
class MathError(Exception):
    pass
```

Wrapping errors

```
>>> class MathError(Exception):
...     pass
...
>>> try:
...     1 / 0
... except ArithmeticError as e:
...     raise MathError("Division impossible") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.MathError: Division impossible
```

Re-raising errors

```
>>> try:
...     1 / 0
... except ArithmeticError:
...     print("Division impossible")
...     raise
...
Division impossible
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

finally

The `finally` keyword is used to ensure that some code is executed regardless of whether an exception has occured or not.

More details in the doc.

```
>>> try:
...     1 / 0
... finally:
...     print("Hello World")
...
Hello World
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

`else`

Just like for loops, the `else` branch is executed if the `try` branch is completed without an error.

Do you have any questions ?

Python Fundamentals

Native Functions

Introduction

Some functions do not require an import, they are always available.

The complete list is available in the Python documentation.

Basic types

```
>>> int(42.0)
42

>>> str()
''
```

Iteration

```
>>> list(enumerate(["a", "b", "c"]))
[(0, 'a'), (1, 'b'), (2, 'c')]

>>> list(zip(["a", "b", "c"], [10, 20, 30]))
[('a', 10), ('b', 20), ('c', 30)]

>>> list(map(float, [1, 2, 3]))
[1.0, 2.0, 3.0]

>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(reversed([1, 2, 3]))
[3, 2, 1]

>>> sorted([3, 1, 2])
[1, 2, 3]
```

Aggregation

```
>>> all([0, 42])
False

>>> any([0, 42])
True

>>> sum([1, 2, 3])
6

>>> max([1, 2, 3])
3

>>> min([1, 2, 3])
1
```

Objects manipulation

```
>>> class A:
...     pass
...
>>> a = A()

>>> setattr(a, "x", 3)
>>> a.x
3

>>> getattr(a, "x")
3

>>> delattr(a, "x")
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
```

Type testing

```
>>> callable(int)
True
>>> callable(42)
False

>>> type(True)
bool

>>> isinstance(True, int)
True
>>> type(True) == int
False

>>> issubclass(bool, int)
True
```

Basic operations

```
>>> abs(-42)
42

>>> divmod(10, 3)
(3, 1)

>>> pow(2, 3)
8

>>> pow(3, 2, 4)
1
```

Do you have any questions ?

Python Software Engineering

Python Software Engineering

Software Environment Management

Introduction

Control your software environment to :

- Make your development environment reproducible
- Control dependencies for production
- Deploy your environment easily on different clouds

Environment Isolation — venv

venv allows Python environments isolation. A virtual environment:

- Does not interact with the system environment
- Can coexist with other virtual environments
- Is faster than a container-based solution
- Is a copy of a base Python distribution

Using venv

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install requests
```

Dependency Control

Goal : precisely describe which libraries (and their versions) are used.

Several good tools :

`pip` + `requirements.txt` Simple list of dependencies

`conda/mamba` Environment management, including native dependencies

`poetry` Application or library definition inspired by Cargo

`uv` Similar to `poetry`, more comprehensive

Using uv

Some useful commands for dependency management:

`uv init`	Creates a Python package skeleton managed by `uv`
`uv add`	Adds a package as a project dependency
`uv remove`	Removes a package from project dependencies
`uv sync --frozen`	Installs dependencies exactly as specified in the `uv.lock` file (automatically managed by `uv`)
`uv export`	Exports the `uv.lock` file to the `requirements.txt` format

Facilitating Deployment

A few platforms accept Python packages for deployment.

Most often, an intermediate step is required before deployment: transforming into a container. It brings:

- Control over native libraries in addition to Python libraries
- Greater robustness if the software runs on multiple OS
- A heavier setup than `venv`
- A more suitable format for production (easy deployment via Kubernetes)

⇒ Dockerization of Python packages using a `Dockerfile`

Dockerfile Example

```
FROM python:3.14-slim-trixie
WORKDIR /app
RUN apt update \
  && apt install -y --no-install-recommends \
  build-essential python3-dev gfortran libgl1-mesa-glx \
  libopenblas-dev liblapack-dev
COPY --from=ghcr.io/astral-sh/uv:latest /uv /uvx /bin/
COPY ./uv.lock ./pyproject.toml ./
RUN uv sync --frozen --no-install-project --no-dev
COPY . .
RUN uv sync --frozen
CMD ["uv", "run", "landscapred"]
```

Multi-Stage `Dockerfile` with uv Example

```
FROM python:3.14-slim-trixie AS base
WORKDIR /app

FROM base AS uv
COPY --from=ghcr.io/astral-sh/uv:0.9.8 /uv /uvx /bin/
COPY uv.lock pyproject.toml /app/
RUN uv export --frozen --no-dev --no-emit-project -o requirements.txt

FROM base
COPY --from=uv /app/requirements.txt /app/requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY ./pyproject.toml ./README.md ./src ./
RUN pip install .
CMD ["uvicorn", "landscape_classifier.api:app", "--host", "0.0.0.0",
     "--port", "80"]
```

Multi-Stage `Dockerfile` with `poetry` Example

```
FROM python:3.14-slim-trixie AS base
WORKDIR /app

FROM base AS poetry
RUN pip install --no-cache-dir 'poetry == 1.8.2'
COPY poetry.lock pyproject.toml /app/
RUN poetry export -o requirements.txt

FROM base
COPY --from=poetry /app/requirements.txt /app/requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY ./pyproject.toml ./README.md ./src ./
RUN pip install .
CMD ["uvicorn", "landscape_classifier.api:app", "--host", "0.0.0.0",
     "--port", "80"]
```

Do you have any questions ?

Python Software Engineering

Software Environment Management

Demonstration

Demonstration

The different systems seen during the demonstration are :

- `pip install -r requirements.txt` and `pip freeze` to freeze versions
- `venv`
- `uv`, the tool I recommend for managing Python environments

Python Software Engineering

Python Code Typing

Introduction

Adding type annotations is probably the greatest improvement one can make to a Python codebase:

- Allows for aggressive software modifications
- Resolves a large number of bugs before integration into the codebase
- Documents the code
- Enables automatic generation (CLI with Typer, for example)

The transition to a typed codebase can be gradual.

Typing Notations

An expression is typed by preceding the type annotation with `:`.

For function return types, we use `->`.

```
def add(a: int, b: int) -> int:
    return a + b
```

Behavior in Python

```
>>> def add(a: int, b: int) -> int:
...     return a + b
...
>>> add("Hello ", "World!")
'Hello World!'
```

Python does not check types at runtime. To verify them, we use **mypy** beforehand :

```
$ mypy type-hints.py
type-hints.py:5: error: Argument 1 to "add" has incompatible type "str";
expected "int"  [arg-type]
type-hints.py:5: error: Argument 2 to "add" has incompatible type "str";
expected "int"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

Best Practices

- Automatic usage in production pipelines (continuous integration)
- Configuration within the development environment
- Gradual typing of an existing codebase if necessary

Configuration

In the `pyproject.toml` file :

```
[tool.mypy]
disallow_untyped_defs = true
ignore_missing_imports = true
strict_optional = true
```

Resources

- Documentation for `mypy`
- Documentation for the `typing` module

Do you have any questions ?

Python Software Engineering

Python Code Typing

Generics

Introduction

Generics allow functions or classes that manipulate arbitrary types to be typed.

```
from collections.abc import Sequence


def first[T](seq: Sequence[T]) -> T:
    return seq[0]
```

Difference with `typing.Any`

A type variable allows linking the type of multiple values, whereas `typing.Any` is a wildcard :

```
from collections.abc import Sequence
from typing import Any


def first_any(seq: Sequence[Any]) -> Any:
    return seq[0]


def first_typevar[T](seq: Sequence[T]) -> T:
    return seq[0]


data = [0, 1]
first_any(data).upper()
first_typevar(data).upper()  # "int" has no attribute "upper"
```

Generics in a Class

```python
class Stack[T]:
    def __init__(self) -> None:
        self.items: list[T] = []

    def push(self, item: T) -> None:
        self.items.append(item)

    def pop(self) -> T:
        return self.items.pop()

    def empty(self) -> bool:
        return not self.items


s: Stack[str] = Stack()
s.push("abc")
s.push(1)  # Argument 1 to "push" of "Stack" has incompatible type "int";
           # expected "str"
```

Do you have any questions ?

Python Software Engineering

Python Code Typing

Protocols

Introduction

Protocols define interface-like structures without relying on object hierarchy.

Instead, they use structural typing.

Structural Subtyping

Structural subtyping is based on the attributes and methods of an object rather than class hierarchy, as in traditional object-oriented programming.

Also known as *Duck Typing*:

If it has two legs and quacks, it's a duck.

Defining a Protocol

```python
from pathlib import Path
from typing import Protocol


class SupportsSave(Protocol):
    def save(self, path: Path) -> None: ...
```

Using a Protocol

```
from collections.abc import Iterable
from pathlib import Path


class Resource:
    ...

    def save(self, path: Path) -> None:
        self._resource.save(path)

    ...


def save_all(items: Iterable[SupportsSave],
             paths: Iterable[Path]) -> None:
    for item, path in zip(items, paths):
        item.save(path)


resources = [Resource(), Resource()]
paths = [Path.home() / "1.bin", Path.home() / "2.bin"]
save_all(resources, paths)
```

Protocols Available in the Standard Library

- In the module `collections.abc`
- In the module `typing`

Alternatives

`abc.ABC`s and informal interfaces can complement or replace protocols depending on the use case, but protocols remain the most precise way to specify interfaces.

Do you have any questions ?

Python Software Engineering

Python Code Typing

Some More Typing Elements

typing.NewType

NewType creates a distinct type from an existing type.

```python
from typing import NewType

# New type definition based on str
UserName = NewType("UserName", str)

# Variable creation with the new type
user_name = UserName("Jean Martin")


def get_first_name(user_name: UserName) -> str:
    return user_name.split(" ")[0]


# Ok
print(get_first_name(user_name))
# mypy error
print(get_first_name("Marie Jeanne"))
```

The new type retains all the functionalities of the class it derives from and has no impact at runtime.

typing.NewType

Warning : NewType does not actually define a new class and therefore cannot be the parent of another class:

```
>>> from typing import NewType
>>>
>>> # New type definition
>>> UserName = NewType('UserName', str)
>>>
>>> class User(UserName):
>>>     pass
TypeError: Cannot subclass an instance of NewType.
```

typing.TYPE_CHECKING

TYPE_CHECKING is a variable that is always `False` at runtime but is `True` when using `mypy` or another static type checking tool.

```
>>> from typing import TYPE_CHECKING
>>> if TYPE_CHECKING:
>>>     # Required for mypy but not during runtime
>>>     from collections.abc import Iterable
>>>
>>> def print_all(iterable: 'Iterable') -> None:
>>>     pass
>>>
>>> print(Iterable)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Iterable' is not defined
```

typing.Literal

`Literal` allows defining a type that accepts a finite list of values.

```
from typing import Literal

type Action = Literal["rock", "paper", "scissors"]
type Winner = Literal["p1", "p2", "draw"]


def rock_paper_scissors(action_p1: Action, action_p2: Action) -> Winner:
    pass
```

typing.overload

`overload` is a decorator that allows defining multiple signatures for the same function.

```
from collections.abc import Sequence
from typing import overload

class MyList[T](Sequence[T]):
    @overload
    def __getitem__(self, index: int) -> T:
        ...
    @overload
    def __getitem__(self, index: slice) -> Sequence[T]:
        ...
    def __getitem__(self, index: int | slice) -> T | Sequence[T]:
        if isinstance(index, int):
            # Return a T here
        elif isinstance(index, slice):
            # Return a sequence of Ts here
        else:
            raise TypeError(...)
```

Do you have any questions ?

Python Software Engineering

Python Code Typing

Python Versions Differences

Typing Collections from the Standard Library

Python < 3.9:

```
from typing import Dict

d: Dict[str, int] = {"a": 1}
```

Python ≥ 3.9:

```
d: dict[str, int] = {"a": 1}
```

Typing Protocols

Python $<$ 3.9:

```
from typing import Iterable


def print_iterable_upper(iterable: Iterable[str]):
    for item in iterable:
        print(item.upper())
```

Python $\geq$ 3.9:

```
from collections.abc import Iterable


def print_iterable_upper(iterable: Iterable[str]):
    for item in iterable:
        print(item.upper())
```

Typing Unions

Python $<$ 3.10:

```
from typing import Optional, Union

d: Union[dict[str, int], None] = None
d: Optional[dict[str, int]] = None
...
d = {"a": 1}
```

Python $\geq$ 3.10:

```
d: dict[str, int] | None = None
...
d = {"a": 1}
```

Typing Generic Types

Python $<$ 3.12:

```
from collections.abc import Sequence
from typing import TypeVar

T = TypeVar("T")


def first(seq: Sequence[T]) -> T:
    return seq[0]
```

Python $\geq$ 3.12:

```
from collections.abc import Sequence


def first[T](seq: Sequence[T]) -> T:
    return seq[0]
```

Do you have any questions ?

Python Software Engineering

Python Linting

Introduction

Goals:

- Compliance with standards within a team
- Minimizing time lost in code reviews
- Producing more maintainable software

Possibilities

The major points that are usually checked:

- Code style
- Patterns that are often bugs
- Missing or incomplete documentation
- Compliance with best practices
- ...

Tools

Common tools:

- `flake8`
- `pylint`
- `black`
- `isort`
- `pydocstyle`

A modern alternative to all these tools : `ruff`.

General Checks

The essential tool is `ruff`. Depending on the rules enabled, it will check:

- Common bugs
- Usage of deprecated features
- Suboptimal code patterns
- Code complexity
- Naming conventions
- Documentation
- ...

Configuration of `ruff`

`ruff` is configured in the `pyproject.toml` file at the root of your project.

```toml
[tool.ruff]
target-version = "py312"

[tool.ruff.lint]
preview = true
select = ["E4", "E7", "E9", "F", "B"]
```

Style Verification

To handle code style, I strongly recommend using `ruff` with formatting on save:

- Automatic formatting (transformation of source code into a syntax tree and then back into source code ⇒ original formatting is ignored)
- Ability to check application across the codebase
- Consistent style
- No more style issues in code reviews!

There are instructions for each editor in the `ruff` documentation.

Usage Recommendations

Recommended: group checks into a single command (e.g., using a Makefile).

- Run these checks before each PR (git hook possible)
- Integrate them into CI
- A PR should not be reviewed until these issues are resolved

Configuration

- Define as a team which points to check
- Regularly analyze the cost/benefit of each verified element
- Add or remove checks based on these analyses

Reproducibility

Verification tools should be managed like dependencies:

- Each colleague should get the same results given the same codebase
- The tools should be easy to install, including in CI

However, be careful not to install these dependencies in production.

Do you have any questions ?

Python Software Engineering

Python Linting

Labs

Labs

Instructions

Python Software Engineering

Python Packages Creation

Introduction

Goals :

- Be able to publish a Python project as a package on PyPI
- Define the required project dependencies and metadata

Python Package

To transform a folder containing Python sources into a package :

- Add an `__init__.py` file in each source folder
- Add metadata (depends on the build tool)

Tools

- `setup.py`, & `setuptools`
- `Pipenv`
- `poetry`
- `hatchling`
- `uv`

I recommend **uv**. A simple, modern, and complete tool.

Structure of a package managed by uv

```
demo-package
├── pyproject.toml
├── README.md
└── src
    └── demo_package
        ├── __init__.py
        └── py.typed
```

Warning

The source folder name is `src`, followed by the project name with `-` replaced by `_`

`pyproject.toml` File

Standard since PEP 518 :

- Defines the package build tool to use
- Contains metadata
- Specifies dependencies, including development and extras
- Adopted across the entire tooling ecosystem

Example of a `pyproject.toml` File

```toml
[project]
authors = [{name = "m09", email = "142691+m09@users.noreply.github.com"}]
dependencies = ["jupyter >= 1, < 2"]
description = "Experiment with landscape classifying."
license = "Apache-2.0"
name = "landscape-classifier"
readme = "README.md"
requires-python = ">= 3.12"
version = "0.1.0"

[dependency-groups]
dev = ["mypy >= 1"]

[project.scripts]
landscape-classifier = "landscape_classifier.cli:main"

[build-system]
build-backend = "hatchling.build"
requires = ["hatchling"]
```

Using uv

Some useful commands for package management:

`uv build` Builds the package distribution

`uv publish` Publishes the package on PyPI

PyPI Repository

Repository for Python projects :

- Usable with `pip`, `poetry`, `Pipenv`, `uv`, ...
- It is not possible to replace the files of a version with new files
- Can be used in CI pipelines

Do you have any questions ?

Python Software Engineering

Python Packages Creation

Labs

Labs

Instructions

Python Software Engineering

Deployment

Introduction

Goals:

- Make the software available to the rest of the company / the public
- Automate the process
- Deploy frequently to avoid the tunnel effect

Deployment Levels

Deployment is usually done at several levels:

Local	Machine of the developer
Development	Machine or cluster for initial deployment tests
Integration	Continuous integration environment
Staging	Exact replica of the production environment
Production	System used by end users

Continuous Delivery and Continuous Deployment

In English, both are referred to as CD (*Continuous Delivery* and *Continuous Deployment*).

Continuous Delivery

- Short software production cycles
- Regular production of deliverables
- Manual deployment

Continuous Deployment

Like continuous delivery, but with regular and automated deployment.

Best Practices

- Use a controlled environment
- Deploy automatically on a Git tag of a certain format
- Use tools like GitHub Actions, Travis, Jenkins, CircleCI, ...

Example Configuration

```yaml
name: CD

on:
  push:
    tags:
      - "v*"

jobs:
  deploy-pypi:
    name: Deploy to PyPI
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4
      - name: Install uv
        uses: astral-sh/setup-uv@v4
      - name: Publish to PyPI
        run: |
          uv build
          uv publish
```

Do you have any questions ?

Python Software Engineering

Command Line Interfaces

Introduction

Goals:

- Create easily manageable tools
- Enable scripting from your Python programs

Mechanisms

- `sys.argv` is an array that contains the arguments passed to your program
- `argparse`, `click`, or `Typer` parse this array for you

`argparse`

To define a CLI with `argparse`, you need to:

- Create a parser
- Call this parser from the program's entrypoint
- Process the parsed arguments

Creating a Parser

Create an `argparse.ArgumentParser` and then add arguments using the `add_argument` method:

```
import argparse

parser = argparse.ArgumentParser(description="Create datasets.")
parser.add_argument("output_dir", help="Path to the directory.")
parser.add_argument("--langs", nargs="+", help="Langs to consider.")
parser.add_argument("--max-size", type=int, help="Maximum size in ko.")
```

Adding an Argument

You can specify many options to `add_argument`. Among them:

- A short option (`"-d"`) and a long one (`"--directory"`)
- A help string (`help="Argument help."`)
- A type (`type=int`)
- An action, for example, to create boolean flags
- A default value

The official documentation is very comprehensive.

Usage

Call the `parse_args` method and then process the retrieved arguments.

```
import argparse

from .dataset import create_csv_dataset_from_dump

parser = argparse.ArgumentParser(description="Create datasets.")
parser.add_argument("output_dir", help="Path to the directory.")
parser.add_argument("--langs", nargs="+", help="Langs to consider.")
parser.add_argument("--max-size", type=int, help="Maximum size in ko.")

args = parser.parse_args()

create_csv_dataset_from_dump(args.output_dir,
                             langs=args.langs,
                             max_size=args.max_size)
```

Multiple Commands

Ability to define multiple commands: create subparsers and then handle them like the main parser:

```
import argparse

parser = argparse.ArgumentParser(description="Create datasets.")
subparsers = parser.add_subparsers(help="Commands", dest="command")

download_parser = subparsers.add_parser("download")
download_parser.add_argument("output_dir")

dataset_parser = subparsers.add_parser("dataset")
dataset_parser.add_argument("dumps_folder_path")
```

Usage

```
args = parser.parse_args()
if args.command == "download":
    downloader = Downloader(args.output_dir)
    downloader.download_all_wiktionaries()
elif args.command == "dataset":
    create_csv_dataset_from_dump(args.dumps_folder_path)
```

Some Perks Provided by `argparse`

- Automatically generated help
- Informative error messages when arguments are incorrect

Do you have any questions ?

Python Software Engineering

Command Line Interfaces

Labs

Labs

Instructions

Important Modules of the Standard Library

Important Modules of the Standard Library

File Manipulation

Introduction

In Python, file manipulation is mainly done using 4 modules:

`shutil`	High-level copy and move operations
`os`	Low-level operations
`pathlib`	Modern replacement for part of **`os`**
`tempfile`	Creation of temporary files and directories

Navigation · `pathlib` module

```
>>> import pathlib

>>> formations_dir = pathlib.Path.home() / "work" / "formations"
>>> formations_dir
PosixPath('/home/mog/work/formations')
>>> str(formations_dir)
'/home/mog/work/formations'

>>> config_file = formations_dir / "config.yml"
>>> config_file.with_suffix(".txt")
PosixPath('/home/mog/work/formations/config.txt')
>>> config_file.name
'config.yml'

>>> for item in formations_dir.iterdir():
...     print(item)
...
/home/mog/work/formations/formation.sty
/home/mog/work/formations/.git
```

Using wildcards

It is possible to use wildcards thanks to the `pathlib.Path.glob` and `pathlib.Path.rglob` functions:

```
>>> import pathlib


>>> home = pathlib.Path.home()
>>> for jpg in home.rglob("*.jpg"):
...     print(jpg)
...
/home/mog/pictures/cat.jpg
/home/mog/pictures/dog.jpg
/home/mog/work/formations/img/python-logo.jpg
```

Rest of the `pathlib` API

See the excellent official documentation to discover the full API.

Copying, moving & deleting files and directories

`shutil.copy` Copy without metadata

`shutil.copy2` Copy with metadata

`shutil.copytree` Copy a directory

`shutil.move` Move a file or directory

`pathlib.Path.unlink` Delete a file

`shutil.rmtree` Delete a directory

Example use of `shutil`

```
import pathlib
import shutil


home = pathlib.Path.home()
formations_dir = home / "formations"
backup_dir = home / "backup-formations"
old_backup_dir = home / "backup-formations-old"

if backup_dir.exists():
    shutil.move(backup_dir, old_backup_dir)

shutil.copytree(formations_dir, backup_dir)

if old_backup_dir.exists():
    shutil.rmtree(old_backup_dir)
```

Temporary files and directories

`tempfile.TemporaryFile` Create a temporary file

`tempfile.NamedTemporaryFile` Create a temporary file whose name is accessible (not only the object)

`tempfile.TemporaryDirectory` Create a temporary directory

All these methods are used within a **`with`** block to ensure that files/directories are removed.

```
import tempfile


with tempfile.TemporaryFile(mode="w+", encoding="utf8") as fh:
    fh.write(template.render())
```

Creating archives

`shutil.make_archive` creates compressed archives when the corresponding modules are available.

Managing permissions, groups, and owners

The `os` module contains all these functions when they are exposed by the operating system.

Do you have any questions ?

Important Modules of the Standard Library

Environment Variables

Introduction

Environment variables are key for:

- Retrieving system configuration
- Easily deploying Python applications
- Providing an alternative interface to a CLI

Retrieving or modifying an environment variable

Two options:

`os.environ` Dictionary with textual values

`os.environb` Dictionary with binary values

Environment variables are retrieved at the moment the `os` module is imported.

Do you have any questions ?

Important Modules of the Standard Library

Standard Streams

Introduction

The input, output, and error standard streams are privileged communication channels for system tools.

Access in Python

Through the `sys` module:

- `sys.stdin`
- `sys.stdout`
- `sys.stderr`

These streams are exposed as files.

Easy handling of `stdin` to create system tools

The `fileinput` module allows you to create tools that work on either:

- Lines from standard input
- Lines contained in the files whose names are given as arguments

$\Rightarrow$ Easy definition of standard UNIX-style tools.

Redirection of standard output and error

The `contextlib` module provides two useful functions:

- `redirect_stdout`
- `redirect_stderr`

Do you have any questions ?

Important Modules of the Standard Library

External Commands

Introduction

The `subprocess` module allows you to:

- execute system commands
- set up pipelines as you would in a terminal
- capture the output of commands instead of simply displaying it
- check the return code

And much more: official documentation

Code example

```
>>> import subprocess
>>> subprocess.run(["grep",
...                 "linux-libc-dev",
...                 "/var/log/apt/history.log"])
...
Upgrade: linux-libc-dev:amd64 (5.15.0-105.115, 5.15.0-106.116)
CompletedProcess(args=['grep',
                       'linux-libc-dev',
                       '/var/log/apt/history.log'],
                 returncode=0)
```

Do you have any questions ?

Important Modules of the Standard Library

Labs

Labs

Instructions

Important Modules of the Standard Library

Regular Expressions

Introduction

Capture and manipulation of strings using patterns.

Example: add a space after each punctuation

"The cat eats,quietly,its food.John is moved."

↓

"The cat eats, quietly, its food. John is moved. "

Possibilities

Once we have defined a pattern, we can:

- delete it
- modify it
- reuse it
- ...

Special characters

- `^` line start (or negation of a character class)
- `$` line end
- `.` wildcard (matches any character)
- `[` opening of a character class
- `]` closing of a character class
- `(` opening of a capture group
- `)` closing of a capture group
- `?` 0 or 1 appearance of a pattern
- `*` 0 to ∞ appearances of a pattern
- `+` 1 to ∞ appearances of a pattern
- `{` opening of a repetition specifier
- `}` closing of a repetition specifier
- `\` escape character or special sequence
- `|` logical or between patterns

re.match

```python
import re

patterns = ["a.*s", "^b", "s$", "a+b", "a{4,5}b", "y*b"]
test_string = "baaabyys"
for pattern in patterns:
    if re.match(pattern, test_string):
        print(f"{pattern:10} matched")
    else:
        print(f"{pattern:10} did not match")
```

```
a.*s       did not match
^b         matched
s$         did not match
a+b        did not match
a{4,5}b    did not match
y*b        matched
```

re.search

```
import re

patterns = ["a.*s", "^b", "s$", "a+b", "a{4,5}b", "y*b"]
test_string = "baaabyys"
for pattern in patterns:
    if re.search(pattern, test_string):
        print(f"{pattern:10} matched")
    else:
        print(f"{pattern:10} did not match")
```

```
a.*s       matched
^b         matched
s$         matched
a+b        matched
a{4,5}b    did not match
y*b        matched
```

re.search

```
import re

pattern = "a.y"
test_string = "baaabyys"
result = re.search(pattern, test_string)
print(f"Capture of characters {result.start()} to {result.end()}")
print(f"Captured group: {result.group()!r}")
```

```
Capture of characters 3 to 6
Captured group: 'aby'
```

re.findall

```
import re


pattern = "b.."
test_string = "baaabyys"
result = re.findall(pattern, test_string)
print(result)
```

```
['baa', 'byy']
```

re.finditer

```
import re

pattern = "b.."
test_string = "baaabyys"
result = re.finditer(pattern, test_string)
for r in result:
    print(f"{r.group()!r} from {r.start()} to {r.end()}")
```

```
'baa' from 0 to 3
'byy' from 4 to 7
```

Special sequences with \

\d digits $\iff$ `[0-9]`

\D non-digits $\iff$ `[^0-9]`

\s whitespace $\iff$ `[ \t\n\r\f\v]`

\S non-whitespace $\iff$ `[^ \t\n\r\f\v]`

\w alpha-numercial $\iff$ `[a-zA-Z0-9_]`

\W non-alpha-numercial $\iff$ `[^a-zA-Z0-9_]`

Also used as an escape character (e.g. `\.` to match a full stop).

Raw strings

\ is already an escape character for regular Python strings.

Raw strings disable this.

```
string = "before\nafter"
raw_string = r"before\nafter"
print(f"String: {string}")
print(f"Raw string: {raw_string}")
```

```
String: before
after
Raw string : before\nafter
```

Raw strings & patterns

```
import re

pattern_raw = "\\\\"
pattern_normal = r"\\"
test_normal = "before\nafter"
test_raw = r"before\nafter"
for pattern in [pattern_normal, pattern_raw]:
    for test in [test_normal, test_raw]:
        if re.search(pattern, test):
            print("matched")
        else:
            print("did not match")
```

```
did not match
matched
did not match
matched
```

Compiling patterns — Flag `re.IGNORECASE` (`re.I`)

Case-insensitive matching.

```
import re


test = "0ab12Ab3491019AB"
pattern = re.compile(r"\d+[ab]")
print(pattern.findall(test))
pattern = re.compile(r"\d+[ab]\D*", re.IGNORECASE)
print(pattern.findall(test))
```

```
['0a']
['0ab', '12Ab', '3491019AB']
```

Compiling patterns — Flag `re.ASCII` (`re.A`)

Force ASCII character sequences.

```
import re


test = "Élève très studieux"
pattern = re.compile(r"\w+")
print(pattern.findall(test))
pattern = re.compile(r"\w+", re.ASCII)
print(pattern.findall(test))
```

```
['Élève', 'très', 'studieux']
['l', 've', 'tr', 's', 'studieux']
```

Compiling patterns — Flag `re.VERBOSE` (`re.X`)

Allow a more readable syntax.

```
import re

charref = re.compile(
    r"""
        &[#]                    # Start of a numerical entity reference
        (
              0[0-7]+           # Octal form
            | [0-9]+            # Decimal form
            | x[0-9a-fA-F]+     # Hexadecimal form
        )
        ;                       # Final semicolon
    """,
    re.VERBOSE,
)

charref = re.compile("&#(0[0-7]+|[0-9]+|x[0-9a-fA-F]+);")
```

Compiling patterns — Other flags

`re.DOTALL` `.` also captures newlines

`re.LOCALE` Uses the locale to define character sequences

`re.MULTILINE` `^` & `$` match line start & line end instead of string start & string end

Capture group

```python
import re


pattern = re.compile(r"([a-zA-Z]+)(\d+)g*")
r = pattern.finditer("A361224g B4012_w44g")
for res in r:
    print(res.group(0), res.group(1), res.group(2))
```

```
A361224g A 361224
B4012 B 4012
w44g w 44
```

Repetition

```
import re


pattern = re.compile(r"(b.d)+")
r = pattern.finditer("abcdbzdbydbhdefgh")
for res in r:
    print(res.group(0), res.group(1), res.span())
```

```
bcdbzdbydbhd bhd (1, 13)
```

Substitution

```
import re


p = re.compile("(blue|red|green)")
print(p.sub("colour", "blue socks and red shoes"))
print(p.sub("colour", "blue socks and red shoes", count=1))
```

```
colour socks and colour shoes
colour socks and red shoes
```

Substitution with group

It's possible to use captured groups in substitutions (and it's extremely powerful):

```
import re


test = '{"first_name":"James","last_name":"Bond"}'
p = re.compile(r'^\{"(\S+)":"(\S+)","(\S+)":"(\S+)"\}$')
print(p.sub(r"Person(\1='\2', \3='\4')", test))
```

```
Person(first_name='James', last_name='Bond')
```

Splitting

```
import re

test = "I love, regular. expressions"
p = re.compile(r"\W+")
print(p.split(test))
```

```
['I', 'love', 'regular', 'expressions']
```

Do you have any questions ?

Important Modules of the Standard Library

Regular Expressions

Labs

Labs

Instructions

Important Modules of the Standard Library

Databases

Introduction

Relational databases fulfill several needs:

- Structuring data
- Decoupling physical and logical data representations
- Providing transactional capabilities

Key Concepts

- Data is represented as a set of tables
- Each table is a two-dimensional array
- Each data point is represented by a tuple of attributes
- Links between tables establish relationships between entities

Table 1 – People

Id	Age	Weight	Height
1	52	75	178
2	34	76	165
3	18	60	170

Table 2 – Activities

Id	Type	Duration	PersonId
1	Bike	23	1
2	Swim	46	1
3	Swim	18	3

Other Important Concepts

- Each record has a primary key that uniquely identifies it without duplication
- Ability to work transactionally: all modifications are performed atomically, or none are performed

Possible Relationships Between Entities

Relationships are mainly distinguished by their cardinality:

- 1 to 1
- 1 to many
- many to 1
- many to many

Some libraries offer nuances, such as whether to cascade delete or not.

SQL Language

SQL (Structured Query Language) allows manipulation of data in a relational database.

It is not strictly necessary to work with databases in Python, as we'll see shortly.

Implementation

To interact with databases, the most used library is SQLAlchemy.

Versatile, simple for straightforward cases, configurable for complex cases.

Connecting to a Database

Using a URI. The URI includes:

- Database type
- Driver
- User
- Password
- Server address
- Database name

The `echo` option enables debug output.

```
from sqlalchemy import create_engine

engine = create_engine(
    "postgresql+psycopg2://user:password@host/database",
    echo=True)
```

Reading Data

```python
from sqlalchemy import select, text

# SQL Style
with engine.begin() as conn:
    result = conn.execute(text("SELECT x, y FROM some_table"))
    for row in result:
        print(f"x: {row.x}  y: {row.y}")

# SQLAchely Core methods style
with engine.begin() as conn:
    result = conn.execute(select(some_table.c.x, some_table.c.y))
```

Writing Data

```
from sqlalchemy import insert, text

# SQL Style
with engine.begin() as conn:
    conn.execute(
        text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
        [{"x": 6, "y": 8}, {"x": 9, "y": 10}],
    )

# SQLAchely Core methods style
with engine.begin() as conn:
    conn.execute(insert(some_table),
                 [{"x": 6, "y": 8}, {"x": 9, "y": 10}])
```

ORM

In addition to being an SQL engine, SQLAlchemy offers an ORM:

- Model described by Python classes
- Objects synchronized between the database and Python code

Creating Tables

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    fullname = Column(String)


Base.metadata.create_all(engine)
```

Inserting with ORM

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    hugo = User(name="hugo", fullname="Hugo Mougard")
    john = User(name="john", fullname="John Doe")

    session.add_all([hugo, john])

    session.commit()
```

Selecting with ORM

```
from sqlalchemy import select

user = session.scalars(select(User).where(User.name == "hugo")).first()
```

Compatibility

SQLAlchemy is compatible with all major DBMS.

Only a subset of features is exposed for some DBMS.

Handling Migrations

To manage database evolution, SQLAlchemy has a dedicated project:

Alembic

Further Resources

SQLAlchemy Tutorial Course

Do you have any questions ?

Important Modules of the Standard Library

Databases

Labs

Labs

Instructions

Python Testing

Python Testing

General Principles

Introduction

Goals:

- Ensure software quality
- Measure regressions with each update
- Refactor with confidence
- Document code with usage examples for each part of the codebase
- Document assumptions made about the code

Test-Driven Development (TDD)

Interesting development approach based on a cycle:

1. Add a unit test that aims to ensure functional requirements are met
2. Verify that the test fails
3. Create the minimal code that solves the test
4. Verify that all tests pass

Different Types of Tests

Functional Tests:

Unit	Verify a small unit of code (function or class)
Integration	Test the combination of medium-sized code units
Regression	Ensure that the level of quality does not decrease
Smoke	Verify critical (and basic) functionality

Non-functional Tests:

Stress	Measure the system's resistance under high load
Recovery	Measure the system's ability to recover
Security	Check for vulnerabilities
...	

Unit Tests

The core of testing: they verify that the basic building blocks work.

- Usually multiple tests per public API function
- Each test ensures that preconditions, postconditions, and invariants are met

Integration Tests

- Verify that multiple subsystems of the system work correctly together
- Very important for making changes to the system with confidence

End-to-End Tests

- Tests the entire system chain
- Replicates real-world use cases as closely as possible

Regression Tests

- Combination of unit and integration tests
- Also include tests for known bugs (1 bug fixed = 1 test)
- Verify that software quality is maintained after a modification

Some Testing Techniques

Mocking	Replace modules or functions with mocks
Generation	Provide many random inputs to a test to detect bugs
Mutation	Mutate the program, and the test suite should fail after the mutation
Contracts	Often based on generation, they specify invariants and pre/post conditions

Testing Techniques — Mocking

- Replace a class, a method, or a module attribute with a fake object
- Allows testing of systems that are not accessible during tests
- Implemented by `unittest.mock` or `pytest.monkeypatching`

```python
def test_get_json(monkeypatch):

    def mock_get(*args, **kwargs):
        return MockResponse()

    monkeypatch.setattr(requests, "get", mock_get)

    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"
```

Testing Techniques — Generation

- Often coupled with property-based testing
- Focuses on discovering pre/post conditions and invariants
- Test case generation is automatic
- Reduces error cases to simple cases

Testing Techniques — Mutation

- Assumption that a good test suite fails on different code
- Performs subtle mutations
- Counts failed and unfinished tests

Testing Techniques — Contracts

- Extension of property-based testing
- Explicitly specifies expectations on functions and classes

```
@deal.pre(lambda *args: all(arg > 0 for arg in args))
def sum_positive(*args):
    return sum(args)
```

Best Practices

- Follow the four-step structure: setup, execution, validation, cleanup
- Create independent tests
- Test only the public API, never implementation details
- Keep tests fast
- Use mutation testing or fuzzing
- Practice inversion of control

Inversion of Control — Coupling

```python
class Simulator:
    def __init__(self):
        self.api_client = APIClient()
        self.api_key = os.environ("API_KEY")

    def simulate(self):
        ...
```

Inversion of Control — Cohesion

Best practice: each part is testable and easily replaceable.

The instantiation logic varies depending on the context of use.

Allows dependency injection.

```
class Simulator:
    def __init__(self, api_client, api_key):
        self.api_client = api_client
        self.api_key = api_key

    def simulate(self):
        ...
```

Setting Up in Python

`unittest`	Testing framework from the standard library
`pytest`	Most popular testing framework
`hypothesis`	Fuzzing, generation strategies
`mutmut`	Mutation testing
`deal`	Contracts

The first three libraries have many extensions.

Do you have any questions ?

Python Testing

`unittest`

Introduction

Goals:

- Have a testing framework similar to JUnit in Java in the standard library
- Manage the 4-phase lifecycle of a test

Principles

- Define test scenarios by creating classes that inherit from `unittest.TestCase`
- Each scenario consists of multiple tests
- Each test is defined with specific `assert...` methods
- Preparation and cleanup at the function level using `setUp` and `tearDown` methods
- Preparation and cleanup at the class level using `setUpClass` and `tearDownClass` methods

Test Methods

A large number of test methods are available.

Execution Order

1. `setUpClass`
2. `setUp`
3. The test function
4. `tearDown`
5. `tearDownClass`

Steps 2–4 are repeated for each test before step 5 is executed.

Example Usage

```
import unittest


def add(a, b):
    return a + b


class AddTests(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.integers = [(1, 2), (20, 22)]

    def test_commutativity(self):
        for a, b in self.integers:
            self.assertEqual(add(a, b), add(b, a))

    def test_associativity(self):
        for a, b in self.integers:
            self.assertEqual(add(add(a, b), b), add(a, add(b, b)))
```

Running the Test Suite

To run the test suite, the most common method is:

```
$ python -m unittest discover
```

It is also possible to specify a file, a class, or a function to execute.

Mocking Module

`unittest` has a very interesting mocking module: `unittest.mock`.

Two important points:

- The `patch` decorator / context manager, which allows temporary replacement of an object
- The `Mock` class, which allows replacement of untestable objects

It is possible to test which methods have been called and how on a `Mock` object.

Do you have any questions ?

Python Testing

`pytest`

Introduction

Goals:

- Simplify the writing of tests compared to `unittest`
- Provide a modular mechanism for setup and cleanup

Usage

`pytest` uses simple `asserts` to create tests.

```
def add(a, b):
    return a + b


def test_add():
    assert add(2, 2) == 4
```

Fixture

A fixture defines a test environment:

- Connection to a test database
- Creating a directory with files to be tested
- ...

Using a Fixture

```python
import pytest


@pytest.fixture
def integers():
    return [(2, 2), (21, 21)]


def add(a, b):
    return a + b


def test_add(integers):
    for a, b in integers:
        assert add(a, b) == add(b, a)
```

Modularity of Fixtures

A fixture can use multiple other fixtures.

A function can use multiple fixtures.

Fixture Memoization

`pytest` defines several levels of fixture calculation:

- Each function (by default)
- Each class
- Each module
- Each package
- For the entire test session

Fixtures Used by Multiple Modules

Fixtures defined in the `conftest.py` file can be used by all modules in the directory.

Compatibility

`pytest` can run test suites defined with `unittest` or `nose`.

Extensions

Many plugins available.

Do you have any questions ?

Labs

Instructions

Hugo Mougard
hugo@mougard.fr