FS12

Week 02
Python Basics II

Mikhail Masyagin
Daniil Devyatkin

Programming paradigms I

- During our previous lessons we were writing code with `if-elif-else` conditions and `for` and `while` loops. Sometimes we used objects. In one example we have created our own `Interval` class.
- We were mostly working in **Imperative Programming** paradigm, **Procedural Programming** paradigm and sometimes in **Object-Oriented Programming** paradigm.
- What does it mean?

Programming paradigms II

- Programming paradigms are a way to classify programming languages and code-writing methodologies based on their features. Languages can be classified into multiple paradigms:
  - **Imperative**, in which the programmer instructs the machine how to change its state (finite state machine theory).
  - **Procedural**, which groups instructions into procedures (functions).
  - **Declarative**, in which the programmer declares the desired result, but not how to compute it.

Programming paradigms III

- **Functional**, in which the desired result is declared as the value of a series of function applications.
- **Logic**, in which the desired result is declared as the answer to a question about a system of facts and rules.
- **Reactive**, n which the desired result is declared with data streams and the propagation of change.

- Today we will mostly talk about four first paradigms: **Imperative**, **Procedural**, **Declarative**, **Functional**.

Imperative Programming I

- **Imperative programming** is a programming paradigm of software that uses statements that change a program's state.
- In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform.
- Imperative programming focuses on describing how a program operates step by step, rather than on high-level descriptions of its expected results.
- Basis: finite state machines and discrete math.

Imperative Programming II

- **Imperative programming** is the oldest one programming paradigm: first computers with all their hex-coded commands were completely imperative.
- It is one of the most common paradigms.
- It includes other paradigms like **Procedural programming** & **Object-Oriented programming**.
- It allows you to get maximal performance most of the time.
- It is hard to **formally verify** imperatively coded software.

Imperative Programming III

- For us as Python developers **Imperative Programming** is the usage of following language constructions:
  - `if-elif-else`;
  - `for` & `for-in`;
  - `while`.

Procedural Programming I

- **Procedural programming** is a programming paradigm, derived from **Imperative Programming**, based on the concept of the procedure call.
- Procedures (functions) (a type of routine or subroutine) simply contain a series of computational steps to be carried out.
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

Procedural Programming II

- **Procedural programming** became popular when people decided to stop writing all the code every time they create a new program.
- It saves a lot of time, improves code readability and allows easily extend and modify existing code.
- For us as Python developers **Procedural Programming** is about writing and calling functions.

Object-Oriented Programming

- **Object-Oriented Programming** is a programming paradigm based on the concept of objects, which can contain data and code.
- The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).
- For us as Python developers **Object-Oriented Programming** is about writing code for new objects (classes) and their usage.
- It was created in the 1960s.
- We will talk about OOP in the next lecture.

Declarative Programming I

- <u>**Declarative Programming**</u> is a programming paradigm - a style of building the structure and elements of computer programs, that expresses the logic of a computation without describing its control flow.
- Common examples: HTML, CSS, SQL.
- Rare examples: formal verification languages like TLA+, CoQ.
- It was created in 1960s.
- Pure <u>**Declarative Programming**</u> is not about Python :(

Declarative Programming II

- SQL query example:

```sql
SELECT * FROM "lecturers" WHERE
"name" = 'Mikhail';
```

- Small HTML example:

```html
<!DOCTYPE html>
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

Functional Programming I

- **Functional Programming** is a programming paradigm where programs are constructed by applying and composing functions.
- It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.
- In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names, passed as arguments, etc.

Functional Programming II

- First **Functional Programming** languages (Lisp) were developed in early 1950s. Interesting fact that until 1970s some manufacturers were producing Lisp-machines: computers with architecture specially optimized for Lisp computations.
- Now we do not have them, because as you could remember, architecture should be as simple as possible.
- It is easy to **formally verify** (ha-ha).

Functional Programming III

- True **<u>Functional Programming</u>** does not support:
  - Loops, because they use recursion (with tail recursion optimization)
  - `` `if-elif-else` ``, because they use pattern-matching.
  - Variables reassignment, because you always have to create new variables.
  - No side-effects (all functions should be pure (not all, ha-ha)).
  - Basis: lambda calculus.

Functional Programming IV

- For us as Python developers **Functional Programming** is the usage of following functions and constructions:
  - `map`, `filter`, `reduce`;
  - `lambda`-functions;
  - `functools` library.

Lambda

- `lambda`-functions are anonymous functions, which you can create directly in your code and assign them to variables. Also you can return lambdas from functions:

  ```
  lambda <args>: <one string logic>
  ```

- Examples:

  ```
  (lambda x, y: x + y)(2, 3)

  t = [(2, 'v'), (1, 'd'), (5, 'a')]
  sorted(t, key = lambda x: x[1])
  > [(5, 'a'), (1, 'd'), (2, 'v')]
  ```

Map

- \`map\` is a function, that applies given function to a collection, and returns collection like input:
  map(<func>, <iterable>…)
- Be careful: \`map\` returns **map object** instead of sequence. You have to directly convert it (!!!)
- Examples:

```
list(map(int, ['1', '2', '3'])) > [1, 2, 3]
set(map(lambda x: x.lower(), {'ABCD', 'S'})) > {'s', 'abcd'}
list(map(lambda x, y: x + y, [1, 2, 3], [4, 5, 6])) > [5, 7, 9]
```

Reduce I

- `reduce` is a function, that applies given function to a collection, transforming it to single value:
  reduce(<func>, <iterable>,<init>)
- Example:

```
from functools import reduce
a = [1, 24, 17, 14, 9, 32, 2]
cond = lambda a, b: a if a > b else b
reduce(cond, a, 0) > 32
```

Reduce II

- "This is actually the one I've always hated most, because, apart from a few examples involving + or *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the reduce() is supposed to do. So in my mind, the applicability of reduce() is pretty much limited to associative operators, and in all other cases it's better to write out the accumulation loop explicitly." (c) BDFL

Filter

- `filter` is a function, that filters a sequence by a given condition:

```
filter(<func>, <iterable>)
```

- Be careful: `filter` returns <u>**filter object**</u> instead of sequence. You have to directly convert it (!!!)
- Example:

```
a = [1, 24, 17, 14, 9, 32, 2]
cond = lambda x: bool((x + 1) % 2)
filter(cond, a) > [24, 14, 32, 2]
```

Modules I

- A module in Python is a file with .py extensions. It just encapsulation method for readability code;
- Example of module import:

```
import math
math.sqrt(4) > 2.0
```

- Sometimes two modules contain functions with same names. To overcome this we can use aliases:

```
from math import sqrt as s
import numpy as np
s(4) > 2.0
```

Modules II

- Concrete function import:

```
from math import sqrt
sqrt(4) > 2.0
```

- Import all (not recommended):

```
from math import *
```

- You can write your own module and import it like built-in modules.

Packages I

- If you want write your own library, you have to learn packages.
- A package in Python is a directory, that includes subdirectories & modules and contains file `__init__.py`.
- Example:

```
helloworld-project
|---- helloworld
|        |-- __init__.py
|        |--core.py
|-- setup.py
```

Packages II

- During package import (`import package`, `from my_cool_lib import package`) only `__init__.py` is imported (everything that is written inside it is performed).
- For example:

```
from ._dict_vectorizer import DV
from ._hash import FeatureHasher as FH


__all__ = ['DV', 'FH']
```

Packages III

- But what is a magic `` `setup.py` `` file in a `` `helloworld-project` ``?
- The `` `setup.py` `` is a special service file for package manager (`` `pip` ``).
- `` `setup.py` `` file can contain:
  - library version;
  - required packages;
  - python version;
  - license.

Packages IV

- `setup.py` example:

```
from setuptools import find_packages,
setup

MAIN_REQUIREMENTS = ["airbyte-
cdk~=0.1",]
TEST_REQUIREMENTS = [ "pytest~=6.2",
"requests-mock~=1.9.3", "pytest-
mock~=3.6.1"]
```

Packages V

```
setup(
    name="source_yahoo_finance",
    description="Yahoo Finance.",
    author="Airbyte",
    author_email="contact@airbyte.io",
    packages=find_packages(),
    install_requires=MAIN_REQUIREMENTS,
    package_data={"": ["*.json"]},
    extras_require={
        "tests": TEST_REQUIREMENTS,
    })
```

Packages VI

- Distribution: a project containing `` `setup.py` `` file can be built via:
  ```
  > python setup.py sdist > lib.tar.gz
  ```
- Then this archive can be installed via:
  ```
  > pip install lib.tar.gz
  ```

__main__ I

- If you run Python's file via Python interpreter, interpreter will run all the code in a file unlike other programming languages (C\C++, Go, Rust), because Python do not have an entry point.

```
> python program.py
```

- When the interpreter runs a Python file as the main program, it sets the `__name__` variable to `"__main__"`;
- If we want to execute some code only if this module is a file of main program, we can do the following:

__main__ II

```
if __name__ == '__main__':
    <some_executable_code>
```

- Recommended way to write ` '__main__' ` logic in Python:

```
def main():
    print('Hello, World!')

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

requirements.txt I

- We install Python packages via `pip` manager.
- If our package requires other external pages, we can list all of them in the `requirements.txt` file in the root of the package.
- When we install package via `pip`, all its dependencies, listed in `requirements.txt` (and their sub-dependencies, listed in their `requirements.txt`’s) are installed automatically.

requirements.txt II

- `requirements.txt` file can be generated via `pipreqs` library.
- Requirements listed in `requirements.txt` file can be installed manually via:

```
pip install -r requirements.txt
```

- `requirements.txt` example:

```
torch==2.2.3
sklearn==0.35
numpy~=1.4
```

I/O & files I

- What can we do with files?
- Operations, that require file opening:
  - **`open`** & **`close`**;
  - **`write`** & **`read`**;
- Operations, that does not require file opening:
  - **`rename`**;
  - **`copy`**;
  - etc.

I/O & files II

- When a file is opened, the OS receives a special f**<u>ile descriptor</u>** (fd), that uniquely determines which file is currently in use.
- In Python interaction with files is carried out through a special abstract file objects.

  Syntax and arguments:

```
fd = open("some.file", "r")
data = fd.read()
print(data)
fd.close()
```

I/O & files III

- Modern way to interact with files:

```
# fd is closed automatically
with open("some.file", "r") as fd:
    data = fd.read()
print(data)
```

I/O & files IV

Mode	Description
“r”	Read only.
“w”	Write only. the contents of the file are deleted, if the file does not exist, a new one is created.
“rb”	Analogue “r” mode for binary format.
“wb”	Analogue “w” mode for binary format.
“r+”	“r” + “w” mode
“a”	For writing, the information is added to the end of the file.
“x”	To write if the file does not exist, otherwise an exception will be thrown.


Finish!!!