Introduction I

Gabriele Pompa
gabriele.pompa@unisi.com

March 23, 2020

Welcome to the *IT for Business and Finance* class 2019/20. This class is going to teach you financial applications using the Python programming language.

Credit: Several contents and examples of this notebook are taken from the book *Python for Finance – Mastering Data-Driven Finance* (2nd edition) by Yves Hilpisch (O'Reilly).

Contents

Resources 1

1 What is Python? **1**
1.1 Modules and Packages 3

2 Why Python is relevant for Finance? **4**

3 Why Python is relevant for you? **5**

Resources:

- *Python for Finance (2nd ed.)*: Sec. 1.The Python Programming Language, 1.Technology in Finance, 1.Python for Finance, 1.Data-Driven Finance

1 What is Python?

Executive Summary from the Python.org website:

```
Python is an interpreted, object-oriented, high-level programming
language with dynamic semantics. Its high-level built in data
structures, combined with dynamic typing and dynamic binding, make
it very attractive for Rapid Application Development, as well as for
use as a scripting or glue language to connect existing components
together. Python's simple, easy to learn syntax emphasizes readability
and therefore reduces the cost of program maintenance. Python supports
modules and packages, which encourages program modularity and code
reuse. The Python interpreter and the extensive standard library
are available in source or binary form without charge for all major
platforms, and can be freely distributed.
```

So many weird words, right? Let's try to clarify a bit:

- *Interpreted*: an interpreted language is a type of programming language for which most of its implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions.
- *Object-oriented*: object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
- *High-level* programming language: a high-level programming language is a programming language with strong abstraction from the details of the computer. In contrast to low-level programming languages, it may use natural language elements, be easier to use, or may automate (or even hide entirely) significant areas of computing systems (e.g. memory management), making the process of developing a program simpler and more understandable than when using a lower-level language. The amount of abstraction provided defines how "high-level" a programming language is.
- *Dynamic semantics*: the *semantics* of a (programming) language refers to the meaning of the languages, as opposed to their form. For compiled languages, *static semantics* essentially include those semantic rules (that is, restrictions on the structure of valid texts) that can be checked at compile time. For example checking that every identifier is declared before it is used or checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name). *Dynamic semantics,* or *execution semantics,* defines how and when the various constructs of a language should produce a program behavior. For example, it may define the strategy by which expressions are evaluated to values or the manner in which control structures conditionally execute statements.
- *Dynamic type (checking)*: a *type system* is a logical system comprising a set of rules that assigns a property called a 'type' to the various constructs of a computer program, such as variables, expressions, functions or modules. *Type checking* is the process of verifying and enforcing the constraints of types. It may occur at compile-time (*static type check* of program's text) or at run-time (*dynamic-type check* of the type safety of the program).
- *Dynamic binding*: *dynamic binding* (also known as *late binding*) is a mechanism by which a computer program waits until runtime to bind the name of a method being called to an actual subroutine (that is, to its machine address).

Ok, so Python is *dynamic*...

Ok, before you start to hate it... Python is not meant to be as pedantic as it seemed above, it's way more *Zen*...

[1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
```

```
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1 Modules and Packages

In Python, a *module* is a Python file (that is, a file ending with `.py`, like `dummy.py`) containing Python statements and definitions. A Python *package*, is a collection of modules in directories that give a package hierarchy.

Functions defined in modules can be accessed prepending the module name to te name of the function, using the *dot* operator `.`as follow: `ModuleName.functionName()`.

Example: let's import the π constant from the `math` module:

[2]:
```
import math
```

now we can access the π constant using the `.` access operator:

[3]:
```
math.pi
```

[3]:
```
3.141592653589793
```

Brief excursus: notice that in Jupyter Notebooks, as this is, there is no need to explicitly type `print` to print on screen an expression. This is because a Jupyter Notebook is a *Read-Eval-Print loop* (REPL) programming environment. That is, it takes single user inputs (i.e., single expressions), evaluates (executes) them, and returns the result to the user (thanks to Matteo Ipri for pointing this out to me).

Of course you get the same result printing it explicitly:

[4]:
```
print(math.pi)
```

```
3.141592653589793
```

The Python Standard Library (STL) comes with basic mathematical functionalities. More specialized functions/calculations need to be *imported* explicitly from dedicated Python modules.

Example: Suppose that we want to compute log(1). The logarithm *log*() is an example of mathematical function not implemented in the Python STL. Invoking directly the `log()` function causes a `NameError`

```
NameError: name 'log' is not defined
```

because the Python Interpreter is not able to resolve the `log` identifier within the namespace of the STL. In other words, there is no such `log()` function defined in the STL.

[5]:
```
# log(1) # un-comment this to induce NameError: name 'log' is not defined
```

The typical module that we use to access mathematical functions in Python is the Numpy module. We import it giving it the *alias* (that is, a different - shorter - name to refer to it in our code) `np`

[6]:
```
import numpy as np
```

Therefore, function `log`, defined in the Numpy module, can be used in our code calling it prepending the `np` to its name

[7]:
```
np.log(1)
```

[7]:
```
0.0
```

which returns 0, as expected… Do you feel the need to refresh basic mathematical functions? This is a good time to do it and Wikipedia is a comprehensive source of information.

2 Why Python is relevant for Finance?

Python can deliver across three dimensions that are crucial for finance professionals:

- *Efficiency:* getting results faster, thus saving costs and time:
- *Productivity:* getting more job done with the same resources (people, assets,…);
- *Quality:* advantages over competing technologies

All these three aspects are addressed by Python because:

- Python has a full stack of scientific packages (e.g. Numpy, Pandas, Scipy,…) which allow users (Analysts, Quants, Strats, Traders,…) to focus on their domain and not/less on technical aspects of the implementation.
- Python covers the end-to-end process from prototyping to production, reducing the need (and the inherent inefficiencies) of separation between the two (e.g. prototype written by the Front-Office, production code developed by IT department).

Example: In the Equity department of the IB division of Deutsche Bank in London, where I used to work, our team of Strats used to manage a proprietary library fully-written in Python, only rarely delegating to the IT department for lower level implementations. This allowed us to react

very efficiently to the requests coming from the Traders. We were able to provide them analysis and analytics with a delivery period of hours/days, instead of weeks (if not more).

3 Why Python is relevant for you?

From efinancialcareers.com:

> If you can't code and you work in finance you risk being pushed out by those who can. Not today. Not tomorrow, but soon.
>
> It's a trend that's visible in decisions by banks like Citi, JPMorgan and Goldman Sachs to encourage their junior traders and investment managers to learn how to code in Python. But it's also a trend that's visible in banks' job ads - which increasingly list coding as a prerequisite - even in jobs far removed from the 'engineering' coal face.
>
> The message is clear: if you can't code, more and more banking jobs will be closed to you soon. There is still time to make amends.

Yes, agree, in finance we tend to be very *friendly* when it comes to recommendations.