Software Engineering – Architectures, Designs, and Code

L01 on Coding

Giancarlo Succi
Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna
g.succi@unibo.it

Location of the material:
https://github.com/GiancarloSucci/UniBo.SE.A2023

Caveat for the Part on Coding – 1

- Here we will reflect on the concepts we saw on architecture and design looking at the code
- We will focus on a concrete example, which could be used also in the overall course
- We will see many examples of code
- Also very detailed, step-by-step examples
- Our focus will be on the software engineering side, though
- That is in understanding why and how people achieved what they achieved
    - not on hacking solutions

Caveat for the Part on Coding

- The most liked programming languages by the instructor
    - 1. C++
    - 2. C
    - 3. Java
    - 4. Haskell
    - 5. SmallTalk
    - 6. . . .
    - . . .
    - n-1. . . .
    - n. Python
- We use Python because it is becoming a standard
- But we start with a mention of the ancestor

Structure of the Part on Coding

- Introduction on Python for our purposes
- Patterns in Python
- Coding in ChatGPT and around

Python for our purposes

- About the language
- Structure of the execution of Python
- Virtual Environment

About the language

- Origin and principles of the language
- Fundamental syntax
- Structure of execution
- String formatting
- Object orientation
- Polymorphism and late binding
- Functions as objects
- Decorators
- Static members
- Structure of the virtual machine

Origin of the language

- The language was conceived by Guido van Rossum at CWI in the '80s, got its first implementation in the early '90, and then was modified and upgraded multiple times
- The latest stable version of Python (at the time of writing these slides) is 3.11.5, released on 5th October 2022
- The language is releases in a "kind of" open source license, the Python Software Foundation License but there are debates about it
- Apparently, Python comes from ABC and SETL (see `https://en.wikipedia.org/wiki/History_of_Python`)
- The instructor sees a strong resemblance of SmallTalk (see the references below)

- **References**:
    - Python vs. Smalltalk: from StackOverflow and from Medium.
    - About Python in wikipedia: `https://en.wikipedia.org/wiki/Python_(programming_language)`, `https://en.wikipedia.org/wiki/History_of_Python` and related pages

Principles of the language

- Multiparadigm (scripting, imperative, object oriented, and functional)
- Dynamically typed
- Garbage collected
- “Batteries included”
- Based on a virtual machine (“kind of” interpreted)
- **References**:
  - About Python in wikipedia: `https://en.wikipedia.org/wiki/Python_(programming_language)`

The Zen of Python (1/2)

- 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.
- **References**:
  - About the Zen of Python in wikipedia: `https://en.wikipedia.org/wiki/Zen_of_Python`

The Zen of Python (2/2)

- By Tim Peters
  - 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!
- **References**:
  - About the Zen of Python in wikipedia: `https://en.wikipedia.org/wiki/Zen_of_Python`

Multiparadigm

- We will now explore Python in its 4 paradigms
  - Scripting
  - Imperative
  - Functional
  - Object Oriented
- With such an approach we will cycle over the syntax and the semantics of the language to get a comprehensive view of it and revising the fundamental principles of software engineering

Python as a scripting language

Scripting

- Install the Python interpreter: multiple options including
  - Mac: see the guidelines in **Pip Upgrade – And How to Update Pip and Python**
  - Windows: like here: **pyenv for Windows**
- Open the Python interpreter
- Try some commands

```
\% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41) [Clang
     13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>> print("Hello World!")
Hello World!
>>> 4+3
7
>>> x = 7+1
>>> x**2
64
>>>
```

Scripting

- Analysing the scripting perspective of Python exposes us to the fundamental control structures
  - variables, if, blocks via indentation, while, for, collections, iterations

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41) [Clang
     13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
>>> ^D
```

Source: `https://docs.python.org/3/tutorial/interpreter.html`

Indentation

- If I forget the indentation ...

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41) [Clang
     13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>> the_world_is_flat = True
>>> if the_world_is_flat:
... print("Be careful not to fall off!")
  File "<stdin>", line 2
    print("Be careful not to fall off!")
    ^
IndentationError: expected an indented block after 'if'
    statement on line 1
>>> ^D
```

Source: `https://docs.python.org/3/tutorial/interpreter.html`

Dynamic Typing

- Typing is dynamic and enforced ...

```
% python3.11
Python 3.11.2 (v3.11.2:878ead1ac1, Feb  7 2023, 10:02:41) [Clang
     13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more
    information.
>>> x = "a string"
>>> x = 1
>>> print(x)
1
>>> y = 1 + "a string"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Source: https://docs.python.org/3/tutorial/introduction.html

Math

- Mathematics follows mostly the usual approaches ...

```
% python3.11
>>> 3+4
7
>>> 6-3.4
2.6
>>> 4*7.0
28.0
>>> 7/3
2.3333333333333335
>>> 7//3
2
>>> 7%3
1
>>> 7**3
343
>>> 3 + _
346
>>>
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Strings (1/2)

- Strings are a bit more peculiar ...

```
>>> x='strings can be single quoted '
>>> y="or double quoted "
>>> x + y + "and  joined with  +  "
'strings can be single quoted or double quoted and joined with +
      '
>>>
>>> z=''' strings can span
... multiple lines using triple
...
... quotes'''
>>> z
' strings can span\nmultiple lines using triple\n\nquotes'
>>> print(z)
 strings can span
multiple lines using triple

quotes
>>>
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Strings (2/2)

- Strings are a bit more peculiar ...

```
>>> w='And I can add \n special chars like in C, which are
    printed using print'
>>> w
'And I can add \n special chars like in C, which are printed
    using print'
>>> print(w)
And I can add
 special chars like in C, which are printed using print
>>> c='concatenation ' 'is done just sequencing strings '
>>> c
'concatenation is done just sequencing strings '
>>> c ' but not using variables'
  File "<stdin>", line 1
    c ' but not using variables'
      ^^^^^^^^^^^^^^^^^^^^^^^^^^

SyntaxError: invalid syntax
>>> 2*' and the n* operator repeats the string n times'
' and the n* operator repeats the string n times and the n*
    operator repeats the string n times'
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Indexing Strings (1/2)

- Strings can be indexed ...

```
>>> e='My Example of Python'
>>> e[0]
'M'
>>> e[19]
'n'
>>> e[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> e[-1]
'n'
>>> e[-19]
'y'
>>> e[-20]
'M'
>>> e[3:5]
'Ex'
>>> e[-4:-1]
'tho'
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Indexing Strings (2/2)

- Ranges are start-included, end-excluded ...

```
>>> e[0:3]
'My '
>>> e[11:]
'of Python'
>>> e[:11]+e[11:]
'My Example of Python'
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

More on Strings

- Strings are immutable

```
>>> e[0]='T'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> w = 'T' + e[1:]
>>> w
'Ty Example of Python'
>>>
```

- Length of a string

```
>>> len(w)
20
>>> len("123")
3
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Collections

- Python does not have arrays
- Python has 4 basic collection types, partly with syntax similar to that of strings:
  - Lists: ordered, mutable collections with duplicates
  - Tuples: ordered, immutable collection with duplicates
  - Sets: unordered collection, without duplications, whose members are immutable
  - Dictionaries: ordered, mutable collections without duplicates

Source: `https://www.w3schools.com/python/python_tuples.asp`

Lists (1/8)

- List are ordered collections of elements, with a syntax resembling that of strings

```
>>> odd = [1, 3, 5, 7, 9]
>>> odd
[1, 3, 5, 7, 9]
>>> odd[0]
1
>>> odd[4]
9
>>> odd[-1]
9
>>> odd[-5]
1
>>> odd[1:3]
[3, 5]
>>> odd[:2]+odd[2:]
[1, 3, 5, 7, 9]
>>> len(odd)
5
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Lists (2/8)

- List are mutable, append-able, slice-able

```
>>> odd[2]=20
>>> odd
[1, 3, 20, 7, 9]
>>> odd.append(11)
>>> odd
[1, 3, 20, 7, 9, 11]
>>> odd[3:5]=[100,200,300]
>>> odd
[1, 3, 20, 100, 200, 300, 11]
>>> odd[1:3] = []
>>> odd
[1, 100, 200, 300, 11]
>>> odd[:] = []
>>> odd
[]
```

Source: `https://docs.python.org/3/tutorial/introduction.html`

Lists (3/8)

- List are nestable and heterogeneous

```
>>> a = [1,2,3]
>>> b = [10,20]
>>> c = [90, 800, -34]
>>> n = [a, b, c]
>>> n
[[1, 2, 3], [10, 20], [90, 800, -34]]
>>> n[2][1]
800
>>> z = [1, "xxx", 3]
>>> z
[1, 'xxx', 3]
>>> z[1]
'xxx'
```

Source: https://docs.python.org/3/tutorial/introduction.html

Lists (4/8)

- Copy of references

```
>>> x = [1, 2, 3, 4]
>>> y = x
>>> y[2]=200
>>> x
[1, 2, 200, 4]
```

Source: https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/

Lists (5/8)

- Shallow copies

```
>>> w = x.copy()
>>> w[1]=350
>>> w
[1, 350, 200, 4]
>>> x
[1, 2, 200, 4]
```

Source: https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/

Lists (6/8)

- Shallow copies (more)

```
>>> z = [11,22,33]
>>> k = [121, 132, 143, 154]
>>> j = [z,k]
>>> j
[[11, 22, 33], [121, 132, 143, 154]]
>>> i = j.copy()
>>> k[1] = 222
>>> i
[[11, 22, 33], [121, 222, 143, 154]]
>>> i[0] = [5, 8, 13]
>>> k[1] = 222
>>> i
[[5, 8, 13], [121, 222, 143, 154]]
>>> j
[[11, 22, 33], [121, 222, 143, 154]]
```

Source: https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/

Lists (7/8)

- Deep copies

```
>>> j
[[11, 22, 33], [121, 222, 143, 154]]
>>> import copy
>>> dc = copy.deepcopy(j)
>>> k[1] = 423
>>> j
[[11, 22, 33], [121, 423, 143, 154]]
>>> dc
[[11, 22, 33], [121, 222, 143, 154]]
```

- Membership

```
>>> aList = [1, 2, 3, 5, 0, 9, 0, 7, 1, 5]
>>> 3 in aList
True
>>> 12 in aList
False
```

Source: `https://www.geeksforgeeks.org/copy-python-deep-copy-shallow-copy/`

Lists (8/8)

- Deletion

```
>>> aList = [1, 2, 3, 5, 0, 9, 0, 7, 1, 5]
>>> aList.pop(2)
3
>>> aList
[1, 2, 5, 0, 9, 0, 7, 1, 5]
>>> aList.pop(-3)
7
>>> aList
[1, 2, 5, 0, 9, 0, 1, 5]
>>> aList.remove(5)
>>> aList
[1, 2, 0, 9, 0, 1, 5]
>>> del aList[1]
>>> aList
[1, 0, 9, 0, 1, 5]
>>> aList.clear()
>>> aList
[]
```

Source: `https://note.nkmk.me/en/python-list-clear-pop-remove-del//`

Tuples (1/5)

- Tuples are ordered, immutable collections of elements, with syntax resembling arrays

```
>>> aTuple=("touple", 2, 9, [10, 2, 3], "start")
>>> aTuple
('touple', 2, 9, [10, 2, 3], 'start')
>>> aTuple[0]
'touple'
>>> aTuple[-1]
'start'
>>> aTuple[2:]
(9, [10, 2, 3], 'start')
>>> aTuple[1] = "tryToChange"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> aTuple.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Source: https://www.w3schools.com/python/python_tuples.asp/

Tuples (2/5)

- Tuples are immutable!

```
>>> anAlias=aTuple
>>> aTuple = aTuple + (1, 2, 3)
>>> aTuple
('touple', 2, 9, [10, 2, 3], 'start', 1, 2, 3)
>>> anAlias
('touple', 2, 9, [10, 2, 3], 'start')
>>> aTuple = aTuple + (3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
>>> aTuple = aTuple + (3,)
>>> aTuple
('touple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3)
>>> len(aTuple)
9
>>> 2 in aTuple
True
>>> 12 not in aTuple
True
```

Source: `https://www.w3schools.com/python/python_tuples.asp/`

Tuples (3/5)

- Tuples are immutable ... but be careful of references!

```
>>> a = [3,4,5]
>>> b=(a,6)
>>> b
([3, 4, 5], 6)
>>> a[0]=1
>>> b
([1, 4, 5], 6)
>>> c = b
>>> a[1]=10
>>> b
([1, 10, 5], 6)
>>> c
([1, 10, 5], 6)
>>> c = b.copy()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'copy'
```

Source: https://www.w3schools.com/python/python_tuples.asp/

Tuples (4/5)

- Tuples are immutable ... but be careful of references!

```
>>> import copy
>>> d = copy.deepcopy(b)
>>> d
([1, 10, 5], 6)
>>> a[2]=8
>>> b
([1, 10, 8], 6)
>>> c
([1, 10, 8], 6)
>>> d
([1, 10, 5], 6)
>>>
```

Source: https://www.w3schools.com/python/python_tuples.asp/

Tuples (5/5)

- Shortcuts and conversions

```
>>> anAlias=aTuple
>>> aTuple += (55,)
>>> aTuple
('touple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3, 55)
>>> anAlias
('touple', 2, 9, [10, 2, 3], 'start', 1, 2, 3, 3)
>>> aList = [1, 2, 3]
>>> aConvertedTuple = tuple(aList)
>>> aList
[1, 2, 3]
>>> aConvertedTuple
(1, 2, 3)
>>> aList[2]=5
>>> aList
[1, 2, 5]
>>> aConvertedTuple
(1, 2, 3)
```

Source: https://www.w3schools.com/python/python_tuples_update.asp/

Sets (1/3)

- Sets are unordered collections without duplicates whose elements cannot change

```
>>> aSet = {1, 2, "red"}
>>> aSet
{1, 2, 'red'}
>>> anotherSet = {3, 4, "green", 4, "green"}
>>> anotherSet
{3, 4, 'green'}
>>> 1 in aSet
True
>>> "green" in aSet
False
>>> "green" in anotherSet
True
>>> oneIsLikeTrue = { 1, True, "green"}
>>> oneIsLikeTrue
{1, 'green'}
>>> len(oneIsLikeTrue)
2
```

Source: https://www.w3schools.com/python/python_sets.asp/

Sets (2/3)

```
>>> aThirdSet = aSet | anotherSet
>>> aThirdSet
{1, 2, 3, 4, 'green', 'red'}
>>> aFourthSet = aThirdSet & {2, 3, 200, 'green'}
>>> aFourthSet
{2, 3, 'green'}
>>>
>>> aSetFromAList = set([9, 8, 7])
>>> aSetFromAList
{8, 9, 7}
>>> aSetFromATuple = set((6, 5, 4))
>>> aSetFromATuple
{4, 5, 6}
>>> aSetFromATuple + aSetFromAList
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Source: https://www.w3schools.com/python/python_sets.asp/

Sets (3/3)

- Mutable elements like lists, the same sets, and dictionaries cannot be part of sets

```
>>> a = {3, 1, 5, 4}
>>> b = [10, 11, 12]
>>> a.add(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> c = (10, 11, 12)
>>> a.add(c)
>>> a
{1, 3, 4, 5, (10, 11, 12)}
>>> d = {3, 4, 1}
>>> a.add(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Source: https://www.w3schools.com/python/python_sets.asp/

Dictionaries (1/2)

- Dictionaries are ordered collections of pair (key:value), indexed by key, where each key can appear only once

```
>>> aDictionary = { "breed":"Dog", "weight":8, "name":"Deimon",
    "age":3}
>>> aDictionary
{'breed': 'Dog', 'weight': 8, 'name': 'Deimon', 'age': 3}
>>> aDictionary["breed"]
'Dog'
>>> len(aDictionary)
4
>>> aDictionary.keys()
dict_keys(['breed', 'weight', 'name', 'age'])
>>> aDictionary.values()
dict_values(['Dog', 8, 'Deimon', 3])
>>> aDictionary.update({"weight":7.5,"color":"blenheim"})
>>> aDictionary
{'breed': 'Dog', 'weight': 7.5, 'name': 'Deimon', 'age': 3, '
    color': 'blenheim'}
```

Source: https://www.w3schools.com/python/python_dictionaries.asp/

Dictionaries (2/2)

- Dictionaries can be largely manipulated

```
>>> aDictionary.popitem()
('color', 'blenheim')
>>> aDictionary
{'breed': 'Dog', 'weight': 7.5, 'name': 'Deimon', 'age': 3}
>>> aDictionary.pop('weight')
7.5
>>> aDictionary
{'breed': 'Dog', 'name': 'Deimon', 'age': 3}
>>> del aDictionary["age"]
>>> aDictionary
{'breed': 'Dog', 'name': 'Deimon'}
```

Source: `https://www.w3schools.com/python/python_dictionaries_remove.asp/`

The range function

- The range function is used to generate lists of numbers for iterations
- It returns a range object, which could then be converted in a list for printing

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(-4, 5))
[-4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(range(2, 20, 3))
[2, 5, 8, 11, 14, 17]
>>> list(range(2, 20, 7))
[2, 9, 16]
>>> list(range(0))
[]
>>> range(4.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

Source: https://thepythonguru.com/python-builtin-functions/range/

Control flow – if

- The if statement is all based on indentation

```
>>> import math
>>> a = 4
>>> b = 5
>>> c = -1
>>> delta = b**2 - 4*a*c
>>> if delta > 0:
...       x1 = (-b - math.sqrt(delta))/(2*a)
...       x2 = (-b + math.sqrt(delta))/(2*a)
... elif delta == 0:
...       x1 = x2 = - b / (2*a)
... else:
...       x1 = complex(-b, math.sqrt(-delta)/(2*a))
...       x2 = complex(-b, math.sqrt(-delta)/(2*a))
...
>>> x1
-1.425390529679106
>>> x2
0.17539052967910607
>>>
```

Source: https://docs.python.org/3/tutorial/controlflow.html/

Control flow – `while`

- The `while` statement is the typical top-tested loop based on indentation

```
>>> i = 5
>>> factorial = 1
>>> while i > 1:
...  factorial *= i
...  i -= 1
... else:
...  print("the result is ",factorial)
...
the result is 120
>>>
```

Source: `https://docs.python.org/3/tutorial/controlflow.html/`

Flow in loops

- Python has three construct to manage the flow in loops:
  - `break`: like in C and Java breaks the loop (the `else` statement is not executed)
  - `continue`: like in C and Java, suspends the current iteration and jumps directly to the next one
  - `pass`: absent in C and Java, moves to the next block
- Now we analyse in details the following snippets

Source: `https://docs.python.org/3/tutorial/controlflow.html/`

break and continue

```
i = 0
print("break")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        break
        print("break: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop break")
i = 0
print("continue")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        continue
        print("continue: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop continue")
```

pass

```
i = 0
print("pass")
while i in range(10):
    i += 1
    if i == 5:
        print("i is 5!")
        pass
        print("pass: I should not get here!")
    print("Standard printout for iteration ",i)
else:
    print("End of loop pass")
```

Output (1/2)

```
break
Standard printout for iteration  1
Standard printout for iteration  2
Standard printout for iteration  3
Standard printout for iteration  4
i is 5!
continue
Standard printout for iteration  1
Standard printout for iteration  2
Standard printout for iteration  3
Standard printout for iteration  4
i is 5!
Standard printout for iteration  6
Standard printout for iteration  7
Standard printout for iteration  8
Standard printout for iteration  9
Standard printout for iteration  10
End of loop continue
```

Output (2/2)

```
pass
Standard printout for iteration  1
Standard printout for iteration  2
Standard printout for iteration  3
Standard printout for iteration  4
i is 5!
pass: I should not get here!
Standard printout for iteration  5
Standard printout for iteration  6
Standard printout for iteration  7
Standard printout for iteration  8
Standard printout for iteration  9
Standard printout for iteration  10
End of loop pass
```

iterator (1/2)

- This is a case when superclasses are very useful
- iterator is a class supporting iterations over collections, that is, referring to a collection of objects, sequentializing and indexing them, and with a method `__next__()` that:
  - returns one by one the elements of the object
  - updaties the state of the iterator so that it always refers to *next* element in the sequence
  - raises an exception StopIteration when there are no more elements to point to
- Please notice the mix of scripting and object orientation

Source: https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__()%20in%203/

iterator (2/2)

```
>>> string = "iter"
>>> stringIterator = iter(string)
>>> next(stringIterator)
'i'
>>> next(stringIterator)
't'
>>> stringIterator.__next__()
'e'
>>> stringIterator.__next__()
'r'
>>> stringIterator.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Source: https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration#:~:text=An%20iterable%20is%20a%20object,__next__()%20in%203/

iterable

- An **iterable** is the base class for anything that can be iterated over
- It has a method `__iter__()` that returns an **iterator**
- This is also what is done by the global function `iter()` (see slide 51)
- Notice the duality global functions and member functions, like for the global function `next()` and member function `__next__()`
- The duality **iterable** – **iterator** is heavily used in `for` loops, list comprehension etc (see later)

```
>>> string = "iter"
>>> stringIterator = string.__iter__()
>>> next(stringIterator)
'i'
```

Source: https://stackoverflow.com/questions/9884132/what-are-iterator-iterable-and-iteration

for loop

- The for loop in Python iterates over an iterator
- At every step in the loop there is an implicit call to the __next__() member function of the iterator
- At the last step there is an implicit catch of the StopIteration exception
- else, break, continue, and pass work like in a while loop

```
>>> string = "iter"
>>> for i in string:
...     print(i)
... else:
...     print("end of string")
...
i
t
e
r
end of string
```

Source: https://wiki.python.org/moin/ForLoop

Command line (1/2)

- Command line arguments work like in Java with `argv`
- In addition there is the function `getopt` that helps parsing the arguments one by one

```
import sys, getopt
print("Argument line: ",sys.argv)
print("Number of arguments: ",len(sys.argv))
i=0
for arg in sys.argv:
        print("argument ",i," : ",arg)
        i += 1
options, arguments = getopt.getopt(sys.argv[1:],"nh:o:")
for opt, val in options:
        if opt=="-h" : print ("Help: ",val)
        elif opt=="-o" : print ("Other: ",val)
        elif opt=="-n" : print ("n means ", end=""); print("\t
            no arguments")
        else : print(opt, " and ", val, " are not acceptable")
print("The other arguments are ", arguments)
```

Source: `https://docs.python.org/3/library/getopt.html`

Command line (2/2)

- The output is:

```
% python3 commandLine.py -o xxx -h help -n a b c
Argument line:  ['commandLine.py', '-o', 'xxx', '-h', 'help', '-
    n', 'a', 'b', 'c']
Number of arguments:  9
argument  0  :  commandLine.py
argument  1  :  -o
argument  2  :  xxx
argument  3  :  -h
argument  4  :  help
argument  5  :  -n
argument  6  :  a
argument  7  :  b
argument  8  :  c
Other:  xxx
Help:  help
n means      no arguments
The other arguments are  ['a', 'b', 'c']
```

Source: `https://docs.python.org/3/library/getopt.html`

Python as an imperative language

Functions (1/4)

- In Python there are functions with parameters and return values

```
# exampleFunction.py
import math
def secondOrderEquation(a, b= 3, c= 0.5):
    delta = b*2 -4*a*c
    x1 = (-b - math.sqrt(delta))/(2*a)
    x2 = (-b + math.sqrt(delta))/(2*a)
    return a, b, c, x1, x2

print(secondOrderEquation(1, 6, 1))
print(secondOrderEquation(1, 6))
print(secondOrderEquation(1))
print(secondOrderEquation(1, c=-1))

% python3 exampleFunction.py
(1, 6, 1, -4.414213562373095, -1.5857864376269049)
(1, 6, 0.5, -4.58113883008419, -1.4188611699158102)
(1, 3, 0.5, -2.5, -0.5)
(1, 3, -1, -3.08113883008419, 0.08113883008418976)
```

Functions (2/4)

- Parameters are all passed by values and values can be references, as in Java
- But there are mutable and immutable objects!
- Functions can be parameters of other functions

```
# higherOrder.py
def apply(f,a):
    return f(a)
def increment(x):
    return x+1
def square(x):
    return x**2
def main():
    x = apply(increment,3)
    y = apply(square,4)
    print(x, y)
if __name__ == "__main__":
    main()
% python3 higherOrder.py
4 16
```

Functions (3/4)

- There are anonymous functions
- They are called `lambda` as they resemble lambda expressions

```
# exampleLambda.py
def apply(f,a):
    return f(a)
def main():
    x = apply(lambda x: x+1,3)
    y = apply(lambda x: x**2,4)
    print(x, y)
if __name__ == "__main__":
    main()
% python3 higherOrder.py
4 16
```

Functions (4/4)

- There is a *kind of* `main` function
- Every module has an internal variable, `__name__`
  - this value is set to `__main__` if the module is the one invoked directly in the command line
- Otherwise, it is set to the *name of the module*, that is, the name of the file without the suffix `.py`
- There is a convention to call a function `main()` as the first function to be executed by a module that is directly called in the command line

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

Nested functions

- Functions can be nested, like in Pascal

```
# nested.py

def outer(x) :
    y = 10
    def inner(z):
        return z + y
    w = inner(x)
    return w

if __name__ == "__main__":
    print(outer(3))

% python3 nested.py
13
```

Scoping rules (1/5)

- LEGB scoping
  1. Local
  2. Enclosing
  3. Global
  4. Builtin
- The keyword `global` declares a variable as global
- The keyword `nonlocal` declares a local variable in an inner scope associated to the outer scope

Scoping rules (2/5)

- Without a `global` or a `nonlocal` declaration:
  - if a variable is initialized with a name already used in an outer scope, then it is treated as a new local variable, which then hides the global one
  - if a variable is used with a name already used in an outer scope, but
    - then if its value is then changed inside the scope
    - an error is raised, as
    - it is treated as a local variable that previously was used without being initialized.

Scoping rules (3/5)

```
# scoping.py
x = 5
def testScope(a):
    y = x + a
    global z
    z = y + 1
    return y + z
def main():
    print (testScope(5) + z)
if __name__ == "__main__" :
    main()
% python3 scoping.py
32
```

Scoping rules (4/5)

```
# nonLocalScoping.py
x = 1; y = 2; w = 3; z = 4
def testScope():
    y = 20; w = 30; k = z;
    def testInnerScope():
        w = 300
        global x
        nonlocal y
        x = 100; y = 200; z = 400
        print("testInnerScope: x=",x,",y=",y,",w=",w,",z=",z)
    testInnerScope()
    print("testScope: x=",x,",y=",y,",w=",w,",z=",z)
def main() :
    testScope()
    print("main: x=",x,", y=",y,", w=",w,", z=",z)
if __name__ == "__main__" :
    main()

% python3.11 nonLocalScoping.py
testInnerScope: x= 100 ,y= 200 ,w= 300 ,z= 400
testScope: x= 100 ,y= 200 ,w= 30 ,z= 4
main: x= 100 ,y= 2 ,w= 3 ,z= 4
```

Scoping rules (5/5)

```
# simpleScopingError.py
x = 1
y = 2
z = 3
w = 0
def testScope():
    global x
    y = 4
    z = 5 + w
    global z
    w = -1
def main() :
    print("in main x =",x,", y =",y)

global z
^^^^^^^^
SyntaxError: name 'z' is assigned to before global declaration

z = 5 + w
        ^
UnboundLocalError: cannot access local variable 'w' where it is
    not associated with a value
```

Default parameters as static (1/3)

- Default parameters are evaluated only once, at the first call
  - they are like static local variables in C
  - normally objects are immutable
  - but when objects are mutable something unexpected may happen

Default parameters as static (2/3)

```
# defaultParametersStatic.py
i = 1
def f(a=[50]):
    global i
    print("Call ", i, "At the beginning of f a is:",a)
    a[0] +=1
    print("At the end of f a is:",a)
    i+=1
def main():
    x = [2]
    print("Before the first call to f x is: ",x)
    f(x)
    print("After the first call to f x is: ",x)
    f()
    print("After the second call to f x is: ",x)
    f()
    print("Before the third call to f x is: ",x)
    f(x)
    print("Before the third call to f x is: ",x)
if __name__=="__main__":
    main()
```

Default parameters as static (3/3)

```
% python3.11 defaultParametersStatic.py
Before the first call to f x is:  [2]
Call  1 At the beginning of f a is: [2]
At the end of f a is: [3]
After the first call to f x is:  [3]
Call  2 At the beginning of f a is: [50]
At the end of f a is: [51]
After the second call to f x is:  [3]
Call  3 At the beginning of f a is: [51]
At the end of f a is: [52]
Before the third call to f x is:  [3]
Call  4 At the beginning of f a is: [3]
At the end of f a is: [4]
Before the third call to f x is:  [4]
```

Modules and Packages (1/n)

- Python structure the code quite like C
- Every file is a module
  - the name of the module is the name of the file without the extension `.py`
  - it is stored in the variable `__name__`
  - such name is changed to `__main__` if the module is the one directly invoked

Source: `https://docs.python.org/3/tutorial/modules.html`

Modules and Packages (2/n)

- To access the module you need to specify the instruction `import amodule`
  - where `amodule.py` is the file where the module is located
  - such file must reside in a location specified by the environmental variable `PYTHONPATH`
  - and any name to use from such module has to be prefixed by the name of the module
  - that is, to use function `goofy()`, the full name `amodule.goofy()` must be specified

Source: `https://docs.python.org/3/tutorial/modules.html`

Modules and Packages (3/n)

- To load the names of the module inside the current namespace the instruction `from amodule import goofy` must be specified
- It is also possible to import all the names inside `amodule.py` with the instruction `from amodule import *`, in which case only the names starting with an _ will not be directly accessible
  - in this way it is possible just to call `goophy()`
- It is also possible to rename an entity or module as
  - `import amodule as unmodulo`, leading to calls like `unmodulo.goophy()` or even
  - `from amodule import goofy as pippo`, leading to calls like `amodule.pippo()`

Source: `https://docs.python.org/3/tutorial/modules.html`

Python as a functional language

o90

Already covered

- Lambda expressions
- Immutable objects
- Unchangeable (by default) global variables inside local scopes

Fundamental syntax

- if-loops
- function calls
- global and local vars

Structure of the execution

- Python is executed on a virtual machine with packages that are loaded at run time
- In this it is similar to Java
- `https://leanpub.com/insidethepythonvirtualmachine/read`

String formatting

- f-syntax

Object orientation

- basics

Polymorphism and late binding

- basics

Functions as objects

- basics

Reflection and Introspection

- basics

Taken from https://betterprogramming.pub/python-reflection-and-introspection-97b348be54d8 and https://betterprogramming.pub/python-reflection-and-introspection-97b348be54d8

Inversion of Control

- basics

Taken from `https://python-dependency-injector.ets-labs.org/introduction/di_in_python.html`

Decorators

- basics

Taken from `https://www.geeksforgeeks.org/decorators-in-python/`

Static members

- basics

Structure of the virtual machine

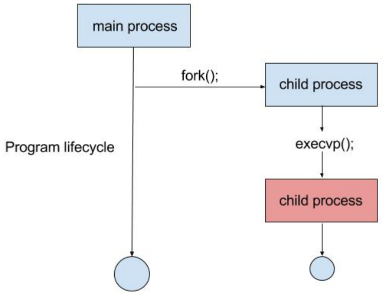
- As said, Python is executed on a virtual machine with packages that are loaded at run time
- In this it is similar to Java
- `https://leanpub.com/insidethepythonvirtualmachine/read`

Virtual Environments

- Often applications needs special configurations of variables to work
- This is well known when we launch scripts
- Typically, scripts invoke shells that set variables and then execute programs
- The variables are modified within the shell, not outside
- In the “outer world” everything remains as is

Ancestors of Virtual Environments (1/2)

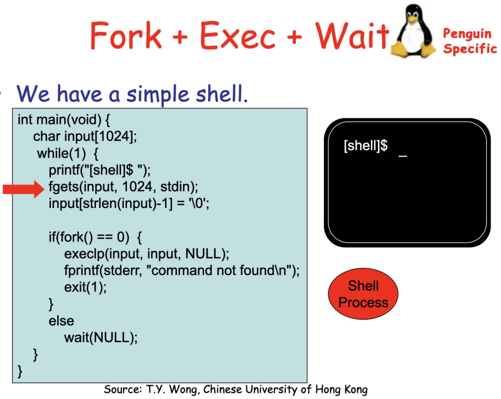
- Creating a new program in the shell
- All the environmental variables established, and all the modifications of variables done in the child process will in general die with the child process



Taken from `https://indradhanush.github.io/blog/writing-a-unix-shell-part-2/`

Ancestors of Virtual Environments

- Code for executing a command in the shell
- Notice all the different kind of `execs`


Fork + Exec + Wait
Penguin Specific
[shell]$ _
Shell Process

- We have a simple shell.

```
int main(void) {
    char input[1024];
     while(1)  {
        printf("[shell]$ ");
        fgets(input, 1024, stdin);
        input[strlen(input)-1] = '\0';

        if(fork() == 0)  {
            execlp(input, input, NULL);
            fprintf(stderr, "command not found\n");
            exit(1);
        }
        else
            wait(NULL);
    }
}
```

Source: T.Y. Wong, Chinese University of Hong Kong

Taken from `https://slideplayer.com/slide/4999027/`

Remember the structure of a Python code

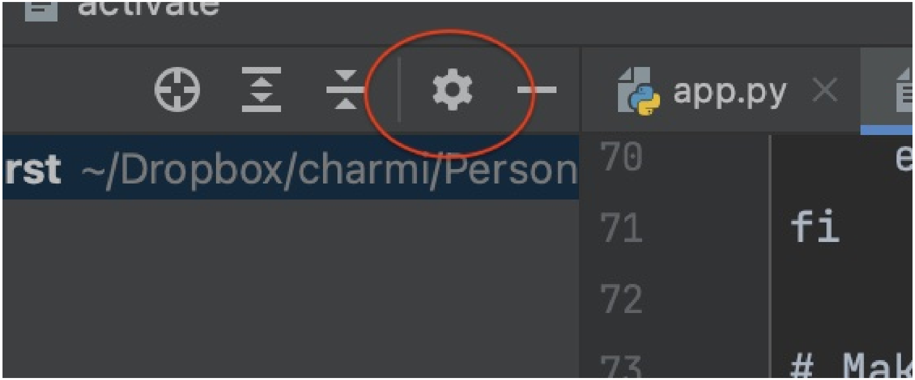
- The “executable” of Python is not a single `a.out` that encloses (in theory) all the required libraries
  - It requires a whole set of packages to load at run time
- Therefore, it is not enough to have a “specialized” environment with suitably set environment variables
  - The “environment” for a program now includes also a set of specific versions of specific packages
- Different programs need different environments

VirtualEnv and VENV

- VirtualEnv `https://sourabhbajaj.com/mac-setup/Python/virtualenv.html`
- venv `https://www.studytonight.com/post/python-virtual-environment-setup-on-mac-osx-easiest-w`

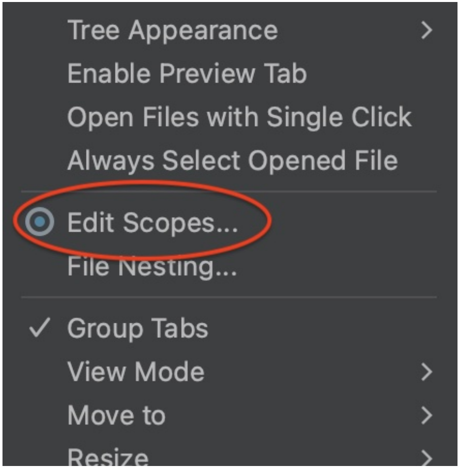
Creating a Virtual Environment in PyCharm

- 1. Click on the settings button



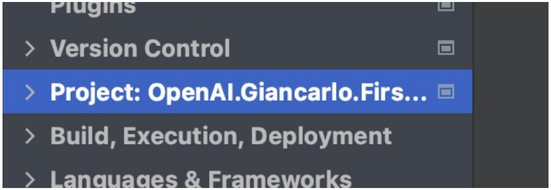
Creating a Virtual Environment in PyCharm

- 2. Edit Scope



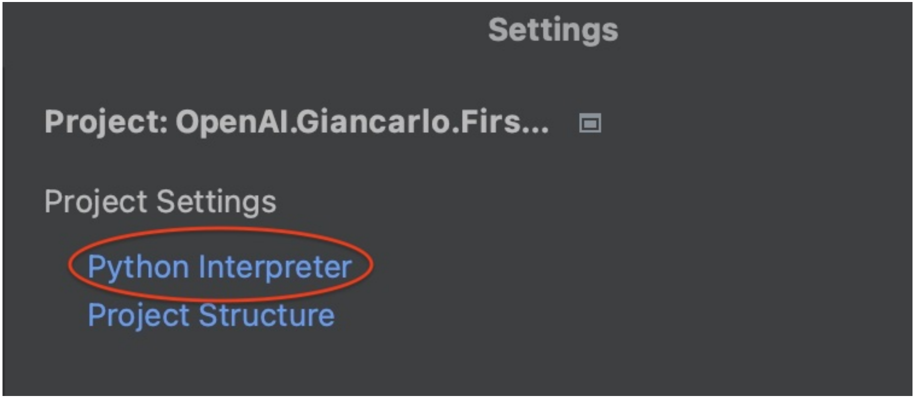
Creating a Virtual Environment in PyCharm

- 3. Select the project


Plugins
Version Control
Project: OpenAI.Giancarlo.Firs...
Build, Execution, Deployment
Languages & Frameworks

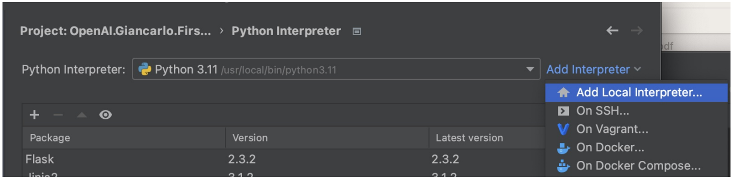
Creating a Virtual Environment in PyCharm

- 4. Select Python Interpreter



Creating a Virtual Environment in PyCharm

- 4. Add Local Interprenter

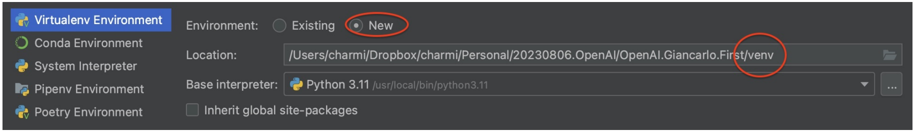

Project: OpenAI.Giancarlo.Firs... › Python Interpreter

Python Interpreter: Python 3.11 /usr/local/bin/python3.11 Add Interpreter

Package	Version	Latest version
Flask	2.3.2	2.3.2

Creating a Virtual Environment in PyCharm

- 5. Select the Virtual Environment and identify an empty directory



Environment: Existing [✓ New]

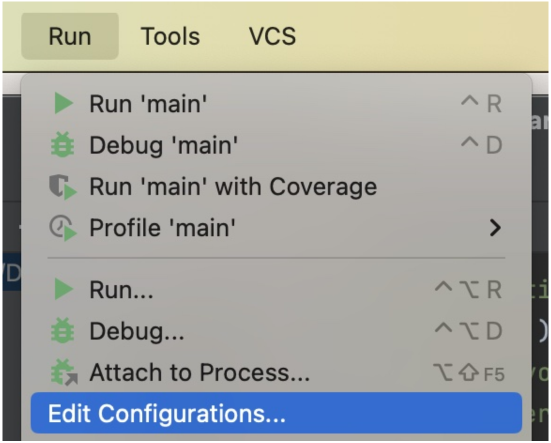
Location: /Users/charmi/Dropbox/charmi/Personal/20230806.OpenAI/OpenAI.Giancarlo.First/venv

Base interpreter: Python 3.11 /usr/local/bin/python3.11

☐ Inherit global site-packages

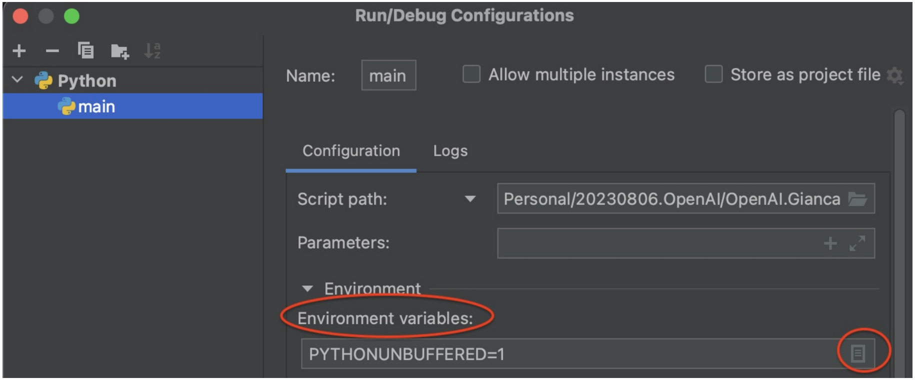
Creating a Virtual Environment in PyCharm

- 6. Set the environmental variables (for using OpenAI key)
  - 6.1 Open the panel



Creating a Virtual Environment in PyCharm

- 6. Set the environmental variables (for using OpenAI key)
  - 6.2 Select the environmental variables



Name: main

Script path: Personal/20230806.OpenAI/OpenAI.Gianca

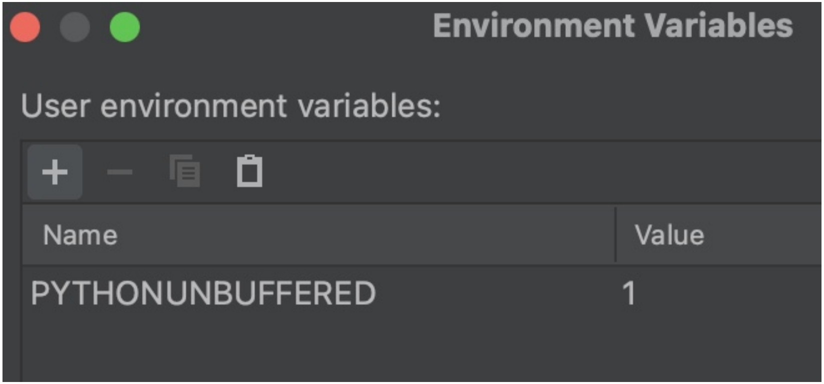
Parameters:

Environment variables:

PYTHONUNBUFFERED=1

Creating a Virtual Environment in PyCharm

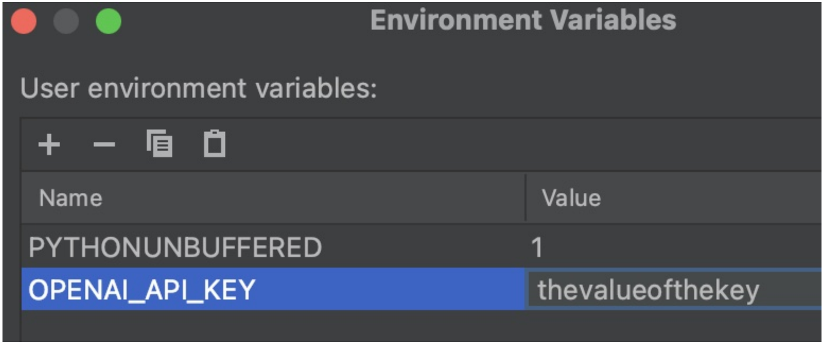
- 6. Set the environmental variables (for using OpenAI key)
    - 6.3 Add an environmental variable



Name	Value
PYTHONUNBUFFERED	1

Creating a Virtual Environment in PyCharm

- 6. Set the environmental variables (for using OpenAI key)
  - 6.4 Set the environmental variable



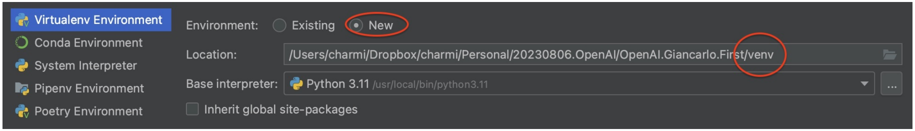
Name	Value
PYTHONUNBUFFERED	1
OPENAI_API_KEY	thevalueofthekey

- Note: The environmental variables can be set in the command line with the usual export mechanism

```
export OPENAI_API_KEY=thevalueofthekey
```

Flask

- To have simple web based applications – inversion of control



Environment: Existing [✓ New]

Location: /Users/charmi/Dropbox/charmi/Personal/20230806.OpenAI/OpenAI.Giancarlo.First/venv

Base interpreter: Python 3.11 /usr/local/bin/python3.11

Inherit global site-packages

Coding in ChatGPT

- Quickstart from the manual
- Understanding the code
- Extending the example

Chat Completion

- Discussion on Chat Completion
- `https://platform.openai.com/docs/guides/gpt/function-calling`
- `https://platform.openai.com/docs/api-reference/chat`

Function Calling

- Discussion on Function Calling
- https://platform.openai.com/docs/guides/gpt/function-calling
- https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb

Questions?

End of the lectures on ChatGPT.