ISTITUTO ITALIANO
DI TECNOLOGIA
DATA SCIENCE AND
COMPUTATION FACILITY

Introduction to Python programming

Sergio Decherchi, PhD

Fondazione Istituto Italiano di Tecnologia, Genoa, Italy

Why should you learn python?

- You need to know a programming language to automatize your activities. Concentrate on science not on running commands.
- It is great for scripting. Bash is ok, but Python is better but also for bigger projects.
- Python is easy.
- Python is free and portable, you can use it in Mac, Windows or Linux
- Python is now considered the "lingua franca" for many disciplines, including: comp. chem. (e.g. rdkit), AI (pytorch, TF, keras, scikitlearn), bioinformatics (nVidia Parabricks), Clinical data (Monai, nVidia FLARE).
- It has a huge community support.
- Several optimized and efficient libraries (e.g. numpy).
- Expected to be time-stable now on.
- It is great for backend and web development.
- PyQT, among others allows to build nice GUI for desktop apps.
- Kivy allows to build cross-platforms (also Android) apps.

Python is not good for some tasks

- Python cannot be used for system-level routines.
- From Python 2.* to 3.* too many changes, spoiled backward compatibility be careful.
- Not officially supported by a company (e.g. Java is supported by Oracle), but many contribute.
- Garbage Collection prevents an efficient management of the memory.
- Python native code is slow. Consider compiling it (e.g. numba) or switch to Groovy/Scala (Java dialect) or Julia.
- Lock (**GIL**). You cannot do multi-core computations (switch to Groovy or Julia).
- OO is supported but not complete (no encapsulation)
- No type checking support
- [Too] many libs dependencies, it is mandatory to manage via conda or containarization the dependencies.

How to install

Linux users:

wget https://repo.anaconda.com/archive/Anaconda3-2023.03-1-Linux-x86_64.sh

- Make it executable via "chmod +x Anaconda3-2023.03-1-Linux-x86_64.sh"
- Run "./Anaconda3-2023.03-1-Linux-x86_64.sh"
- conda install numpy
- conda install matplotlib

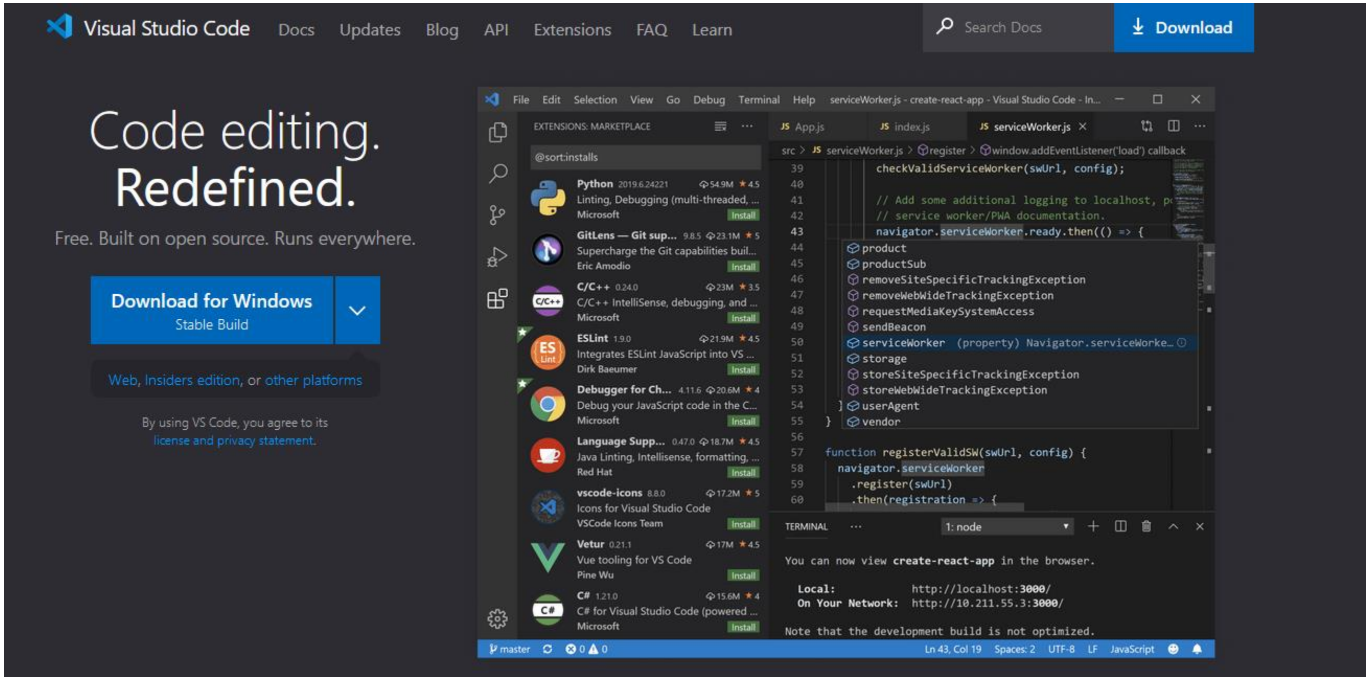
Windows users:

curl –O https://repo.anaconda.com/archive/Anaconda3-2023.03-1-Windows-x86_64.exe

Integrated Development Environment

You can edit with any text editor.
But it is suggested to use one with, at least, syntax highlighting (e.g. Notepad++ in windows, vim in Linux)

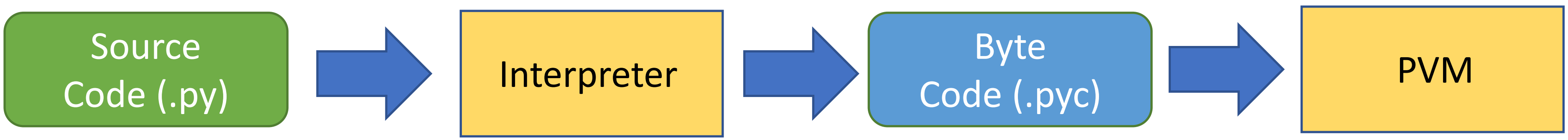
General suggestion: Visual Studio Code, (also integrates with git versioning management)



My first app

- To run the app:

python <u>myapp.py</u>



- To run interactively:

python

To exit type: "exit(0)"

How it looks like?

myapp.py

```
import sys
print("Hi! I am your first app")
print("I run in the following platform:")
myPlat = sys.platform
# print my platform
'''
multiline
Comment
'''
print(myPlat)
```

Import a module

Variable assignment

Comments

Function call

Python data types

Modules contains variables, data structures, functions, classes.

Object type	Example literals/creation
Numbers	`1234`, `3.1415`, `3+4j`, `Decimal`, `Fraction`
Strings	`'spam'`, `"guido's"`, `b'a\x01c'`
Lists	`[1, [2, 'three'], 4]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`
Tuples	`(1, 'spam', 4, 'U')`
Files	`myfile = open('eggs', 'r')`
Sets	`set('abc')`, `{'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes (Part IV, Part V, Part VI)
Implementation-related types	Compiled code, stack tracebacks (Part IV, Part VII)

Numbers

```
a = 4
b = 2
print(a+b)                  6
print(a*b)                  8
print(a/b)                  2.0
print(a//3)                 1
print(a//3.0)               1.0
print(a/3)                  1.3333333333333333
print(a/3.0)                1.3333333333333333
print(b**100)               1267650600228229401496703205376
print("%.2f"%(a/3.0))       1.33
```

Numbers

```
import math as ma

print(ma.sqrt(a))                    2.0
print(ma.sin(ma.pi))                 1.2246467991473532e-16
print("%.4f"%ma.sin(ma.pi))          0.0000

import random as rand

mu = 0.0;
sigma = 1.0;
val = rand.gauss(mu,sigma)
print(val)                          -0.3032748153091l027
```

Strings

```python
myString = "This 'is' a string"
myString2 = 'This "is" a string'
print(myString)
print(len(myString))
print(myString[len(myString)-1])
print(myString[0])
print(myString[1])
print(myString[-1])
print(myString[-1].capitalize())
print(myString.endswith('string'))
print(myString.endswith('g'))
```

```
This 'is' a string
18
g
T
h
g
G
True
True
```

Strings

```
mySString2 = ' and also this one'
concat = myString+myString2
print(myString[0:])
print(myString[0::])
print(myString[:])
print(myString[0:len(myString):1])
print(myString[0:3])
print(myString[0:-1])
print(myString[0:len(myString):2])
print(concat)
```

```
This 'is' a string
This 'is' a string
This 'is' a string
This 'is' a string
Thi
This 'is' a strin
Ti i'asrn
This 'is' a string and also this one
```

Strings are immutable!

```python
myNewString = 't'+myString[1:]
print(myNewString)
myString[0]='t'
```

```
this 'is' a string
Traceback (most recent call last):
  File "c:\Users\sdecherchi\Documents\Ricerca\Lezioni\Python6h+4ex\myapp.py", line 68, in <module>
    myString[0]='t'
    ~~~~~~~~^^^
```

Strings cannot change. You have to allocate them again. This is easy but what happens if you have many big strings?

Converting numbers to strings

`print(a)`	4
`print(b)`	2
`print(str(a))`	4
`print(str(b))`	2
`print(a+b)`	6
`print(str(a)+str(b))`	42

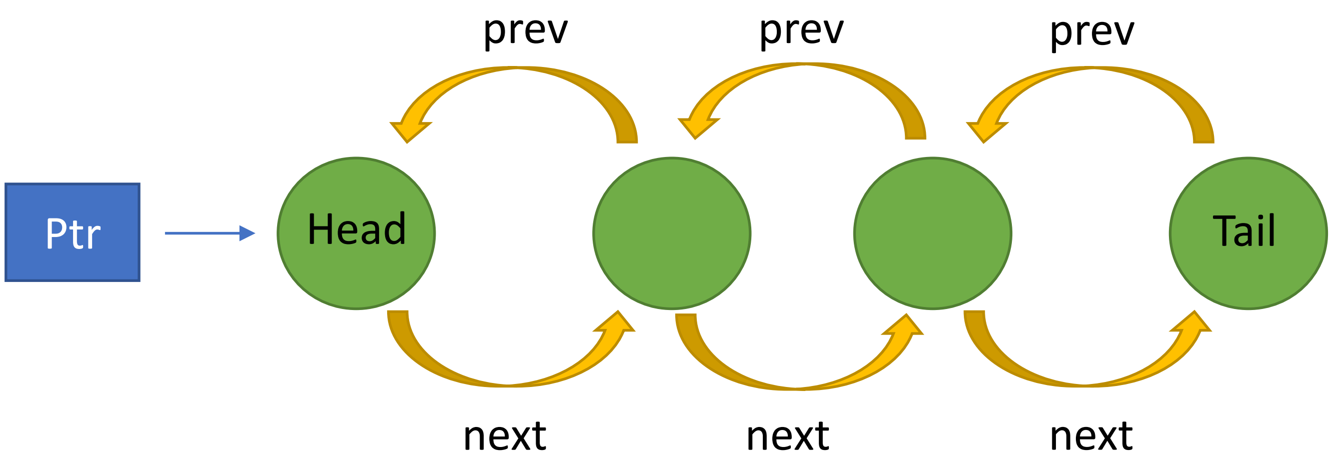
Strings methods

```
myString3 = myString+"  "
print("'"+myString3+"'")
print(myString3.upper())
print("first T at pos %d"%myString3.find('T'))
print(myString3.split(' '))
print(myString3.isalpha())
print(myString3.replace('string','cat'))
print(myString3.rstrip().split(' '))
print('My string is "%s"'%myString3)
print('My strings are "{0}" and "{1}"'.format(myString,myString3))
```

```
'This 'is' a string  '
THIS 'IS' A STRING
first T at pos 0
['This', "'is'", 'a', 'string', '', '']
False
This 'is' a cat
['This', "'is'", 'a', 'string']
My string is "This 'is' a string  "
My strings are "This 'is' a string" and "This 'is' a string  "
```

Lists

- Lists are a concatenation of, possibly hybrid objects.
- Lists are mutable
- Lists can have any size
- Lists can append, remove, find objects
- Lists are memory parsimonious, can be expanded at runtime, but they are sparse, not friendly for calculus!
- Fast insertion, fast deletion (head, tail), good search speed.

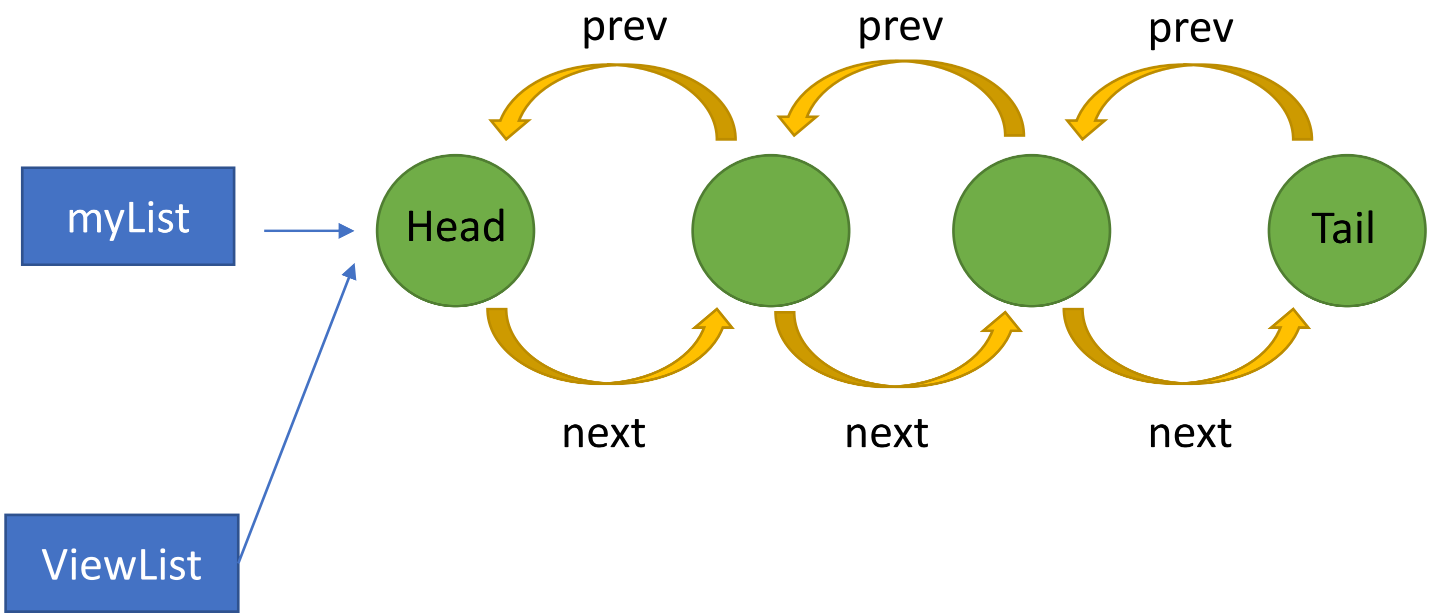


Lists in python

```python
myEmptyList = list()
myList = [0,1,2]
# lists are hybrid
myHybridList = [0,'ciao',1.5]
# lists are mutable
myHybridList[1]='ciao!'
# can be sliced, this allocates new memory
newList = myList[0:2]
# this does change it!
viewList = myList
viewList[0]=20
# this does not change the original list
newList[0]=10

print(len(myEmptyList))
print(myList)
print(myHybridList)
print(myList[0])
print(viewList[0])
print(newList[0])
print(myList[0:2])
```

```
0
[20, 1, 2]
[0, 'ciao!', 1.5]
20
20
10
[20, 1]
```

Pointers



Lists in python

```
# concatenation
myConcat = myList+newList
myView = myConcat

print(myConcat)                    [20, 1, 2, 10, 1]

# removal by index
removedVal = myView.pop(2)
print(removedVal)                  2
print(myConcat)                    [20, 1, 10, 1]
print(myView)                      [20, 1, 10, 1]

# remove by value
myView.remove(20)

print(myView)                      [1, 10, 1]
```

Lists in python

```
myEmptyList.append(5)
myEmptyList.append(10)
myEmptyList.append(1)
myNoMoreEmptyList = myEmptyList
print(myNoMoreEmptyList)

myEmptyList.sort()
print(myEmptyList)
myEmptyList.reverse()
print(myEmptyList)
```

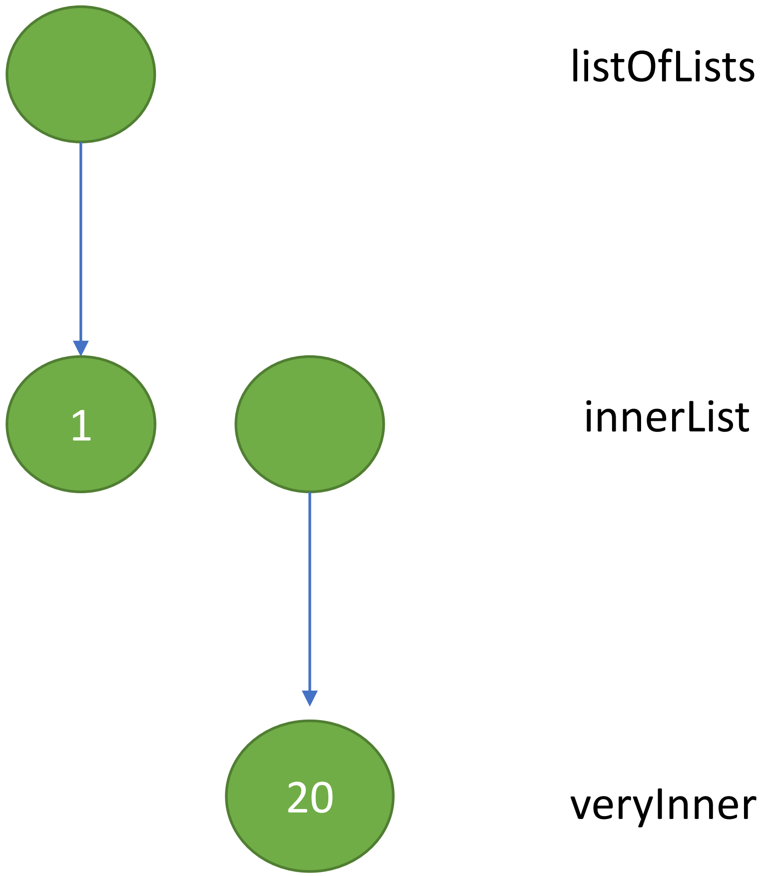
```
[5, 10, 1]
[1, 5, 10]
[10, 5, 1]
```

Nested lists

Python allows for nesting data structures, including lists

```
# nesting

listOfLists = list()
innerList = list()
listOfLists.append(innerList)
innerList.append(1)
print(listOfLists[0][0])

veryInner = [20]
innerList.append(veryInner)
print(listOfLists[0][1][0])

# we can build matrices and tensors in principle (yet very inefficient)
```

Nested lists



List comprehension

```
xx = -1
# list comprehension

numbers = [1, 2, 3, 4, 5]
squared_numbers = [xx**2 for xx in numbers]
print(squared_numbers)

print("Comprehension is not like 'for', does not leek variables xx = %d"%xx)

squared_numbers = []
for xx in numbers:
    squared_numbers.append(xx**2)
print(squared_numbers)
print("For loops leek variables xx = %d"%xx)

# they can be nested
squared_numbers = [[xx**2 for xx in numbers] for xx in numbers]
print(squared_numbers)
```

Tuples

Tuples are immutable sequences in Python, similar to lists, but their elements cannot be modified once created.

- Tuples are created using parentheses ( ) or the tuple() constructor.
- Elements in a tuple are accessed using indexing.
- Tuple Packing and Unpacking: multiple values can be assigned to a tuple in one step.
- Immutable Nature: attempting to modify a tuple results in an error.
- Tuples support common operations like concatenation and repetition.
- Tuples are useful for:
    - Representing fixed collections of related values.
    - Returning multiple values from a function.
    - Immutable keys in dictionaries.
- Benefits of Tuples:
    - Immutable nature ensures data integrity and security.
    - Tuples are hashable and can be used as dictionary keys.
    - Tuples provide a way to group related data without modification.

Tuples

```python
#### tuples

# creation
my_tuple = (1, 2, 3)
another_tuple = tuple([4, 5, 6])

# access
print(my_tuple[0])

my_tuple = (1, 2, 3) # Packing
a, b, c = my_tuple # Unpacking

print(b)

# tuples are immutable
my_tuple[0] = 10
```

```python
# manipulation
tuple1 = (1, 2)
tuple2 = (3, 4)
concatenated_tuple = tuple1 + tuple2 # Output: (1, 2, 3, 4)
repeated_tuple = tuple1 * 3 # Output: (1, 2, 1, 2, 1, 2)

print(concatenated_tuple)
print(repeated_tuple)
```

Dictionaries (Hashmaps)

```
### dictionaries

# Empty dictionary
empty_dict = {}

# Dictionary with initial values
student = {"name": "John", "age": 25, "grade": "A"}

print(student)
```

```
{'name': 'John', 'age': 25, 'grade': 'A'}
```

```
### accessing a dictionary

student = {"name": "John", "age": 25, "grade": "A"}

# Accessing value using key
print(student["name"])

# Using get() method (handles key not found gracefully)
print(student.get("grade"))


# Accessing non-existing key (returns None using get())
print(student.get("address"))
```

```
## checking key existence

student = {"name": "John", "age": 25}

# Using 'in' keyword
if "age" in student:
    print("Age is present")

# Using get() method
if student.get("grade") is not None:
    print("Grade is present")
```

Dictionaries (Hashmaps)

```python
### modifying a dictionary

student = {"name": "John", "age": 25, "grade": "A"}

# Updating value
student["age"] = 26

# Adding a new key-value pair
student["address"] = "123 Main Street"

# Removing a key-value pair
del student["grade"]

print(student)
```

```python
## iterating over a dictionary

student = {"name": "John", "age": 25, "grade": "A"}

# Iterating over keys
for key in student:
    print(key)

# Iterating over values
for value in student.values():
    print(value)

# Iterating over key-value pairs
for key, value in student.items():
    print(key, value)
```

There are also OrderedDicts which preserve the ordering of insertion of the keys

Sets

Sets are unordered collections of unique elements in Python. Sets are mutable, but individual elements must be immutable.

Features:

- Sets are created using curly braces { } or the set() constructor.
- Sets support various operations such as union, intersection, difference, and more.
- Adding and Removing Elements:
- Checking if an element exists in a set using the "in" operator
- Determining the size of a set using the
- Iterating over elements of a set using a loop.

Use Cases:

- Removing duplicates from a list or sequence.
- Testing membership and uniqueness of elements.
- Mathematical operations like union, intersection, etc.

Benefits of Sets:

- Ensures uniqueness of elements.
- Provides efficient membership testing.
- Supports mathematical set operations.

Sets

```python
#### sets

# initialization
my_set = {1, 2, 3}
another_set = set([4, 5, 6])
```

✓ 0.0s

```python
# set operators

set1 = {1, 2, 3}
set2 = {2, 3, 4}
union_set = set1 | set2 # Output: {1, 2, 3, 4}
intersection_set = set1 & set2 # Output: {2, 3}
difference_set = set1 - set2 # Output: {1}

print(union_set)
print(intersection_set)
print(difference_set)
```

If statements

```python
# if statements

a = 5
b = 3

# function evaluation
def myFun():
    print('myFun called')
    return 1

if (a>=5):
    print('a more or equals 5')
if (a==5):
    print('a equals 5')

print('\nsecond test')

if (a>=5):
    print('a more or equals 5')
elif(a==5):
    print('a equals 5')


print('\nnested testing')
if (a>=5):
    print('a more or equals 5')
    if (a==5):
        print('a equals 5')
```

If statements

```
# this produces error (in C does not!)
#if (a=5 and b=3):
#    print('a=5 & b=3')

print('\nor operator')
if (a==5 or b==6):
    print('one of the two is true')

print('\nnested operators')
if ((a==5 and b==3) or 3==3):
    print('one of the two is true')

print('\nfunction evaluation has to be done')
if (a==5 and myFun()==1):
    print('function returns what expected')

print('\nfunction evaluation can be skipped')
if (a==5 or myFun()==1):
    print('function gets skipped')

print('\nfunction evaluation could have been skipped but the interpreter is silly')
if (myFun()==1 or a==5):
    print('function gets skipped')

# a switch/case exists "match" (from python > 3.10 ! so not so portable)
```

For loops

```
start = 4
end = 14
step = 2

for i in range(start,end,step):
    print(i)

print('range separated')
myRange = range(start,end,step)
print(myRange)

for i in myRange:
    print(i)

print('range to list')
asList = list(myRange)

for i in asList:
    print(i)
```

For loops

Scoping in Python 'for' loops

Asked 12 years, 9 months ago Modified 6 months ago Viewed 139k times

253

I'm not asking about Python's scoping rules; I understand generally *how* scoping works in Python for loops. My question is *why* the design decisions were made in this way. For example (no pun intended):

```
for foo in xrange(10):
    bar = 2
print(foo, bar)
```

The above will print (9,2).

This strikes me as weird: 'foo' is really just controlling the loop, and 'bar' was defined inside the loop. I can understand why it might be necessary for 'bar' to be accessible outside the loop (otherwise, for loops would have very limited functionality). What I don't understand is why it is necessary for the control variable to remain in scope after the loop exits. In my experience, it simply clutters the global namespace and makes it harder to track down errors that would be caught by interpreters in other languages.

While loops

```python
i = 0
end = 20
trueend = 10

while (i<end):
    print(i)

    # update here is explicit
    i = i+1

    # ... and fully flexible
    if (i==5):
        i = i+2

    if (i==trueend):
        break

print('forever')
i = 0
while (True):
    print('and ever...')
    if (i==trueend):
        break
    i=i+1
```

Functions = Entia non sunt multiplicanda praeter necessitatem

Functions are reusable blocks of code that perform a specific task. They help in organizing code, improving code reusability, and enhancing readability.

Functions are defined using the **def** keyword followed by the function name and parentheses.
Syntax: def function_name(parameters):

Function Call:
Call a function by using its name followed by parentheses.

Return Statement:
Functions can return a value using the return statement.

Function Parameters:
Functions can accept parameters to pass values for computation.
Parameters can have default values or be passed as positional or keyword arguments.

Functions

Scope of Variables:
Variables defined inside a function have local scope and are accessible only within the function.
Variables defined outside any function have global scope and can be accessed anywhere in the program.

Benefits of Functions:
Code reusability: Functions allow you to write reusable code blocks and avoid errors
Modularity: Functions help in breaking down complex tasks into smaller, manageable parts.
Readability: Functions enhance code readability and maintainability.

Use Cases:
Performing repetitive tasks.
Implementing algorithms or computations.
Encapsulating a specific functionality.

Function Documentation:
It is good practice to provide a docstring that describes the purpose and usage of the function.
Docstrings help in code documentation and can be accessed using the help() function.

Functions, code examples: simple, nested, recursion, function pointers, varargs, doc

```
#### functions

def mul_add(a,b,c):
    return a*b+c

def div_add(a,b,c):
    # returning an error code
    # +1 error
    if (b==0):
        return 0,+1
    # 0 no error
    return a/b+c,0

a = 1
b = 2
c = 3
d = mul_add(a,b,c)
print(d)

d,error = div_add(a,b,c)
if (not error):
    print(f'division ok {d}')

d,error = div_add(a,0,c)
if (error):
    print(f'error, division by 0')
```

Getting help for a method

```
help(myString.replace)
help(myString.__class__.replace)
help(str.replace)
```

```
Help on method_descriptor:

replace(self, old, new, count=-1, /)
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences are
    replaced.
```

Functions, lambdas

Alonzo Church formalized lambda calculus, a language based on pure abstraction, in the 1930s. Lambda calculus can encode any computation, it is Turing complete, but contrary to the concept of a Turing machine, it does not keep any state.

Functional languages get their origin in mathematical logic and lambda calculus, while imperative programming languages embrace the state-based model of computation invented by Alan Turing. The two models of computation, lambda calculus and Turing machines, can be translated into each another. This equivalence is known as the Church-Turing hypothesis.

In python they are "anonymous functions"

When useful? When you can dispose quickly that function without a strong code-overall semantic

They are used in functional programming, python has some support for that. Useful for parallelization

Files handling

File handling is an essential aspect of programming for reading from and writing to files. Python provides built-in functions and methods for file operations:

- Opening a file: it allows to start handling a file, it requires a file name and mode.
- Reading from a file: use the read() or readline() method to read data from a file.
- Writing to a file: use the write() method to write data to a file.
- Closing a file: after reading from or writing to a file, it's essential to close it to let know the OS that other processes can work on it (close()).

File Modes:

r: Read mode (default). Opens a file for reading.
w: Write mode. Creates a new file for writing. Overwrites the file if it already exists.
a: Append mode. Opens a file for appending data. Creates a new file if it doesn't exist.
x: Exclusive creation mode. Creates a new file for writing, but fails if the file already exists.

Files

```
### files handling
fileName1 = 'data.txt'
f = open(fileName1,'w')

f.write('Write a line \n')
f.write('#I am a comment line\n
f.write('Write a second line\n'

f.close()

# this raises an error
fileName2 = 'data2.txt'
#f = open(fileName2,'r')

# we will see later..
#try:
#    f = open('data2.txt','r')
#except Exception as e:
#    t,text = e.args
#    print(text)
#except FileNotFoundError:
#    print('File not found')
```

Files – "with" clause

```python
# files with 'with'

fileName1 = 'data.txt'

# Open and read a file using 'with' statement
# a file is a 'context manager' compatible object
# it manages reosources correctly, giving a scope
with open(fileName1, "r") as file:
    content = file.read()
    print(content)

# The file will be automatically closed outside the 'with' block
```

Modules

Modules in Python are files containing Python code that define functions, classes, and variables.

Benefits of Using Modules:

- Code Reusability: modules allow you to reuse code across different projects and avoid writing the same code multiple times.
- Organized Structure: ,modules provide a logical structure for organizing related code, making it easier to navigate and understand.
- Namespace Isolation: modules have their own namespace, preventing naming conflicts between variables and functions with the same name in different modules.
- Encapsulation: Modules encapsulate related code, making it easier to manage and modify without affecting other parts of the codebase.

Classes

Classes are a fundamental concept in object-oriented programming (OOP) and provide a way to define new data types in Python.

Classes encapsulate data (<u>attributes</u>) and behavior (<u>methods</u>) into a single unit, allowing for code organization and reusability. Classes map concepts.

Some key concepts:

- Classes definition
- Object Instantiation
- Constructors
- The 'self' object
- Static methods and variables
- Operators overloading
- Class inheritance
- Class composition

Enums

Enums are a way to create named constants in Python. Enums define a set of named values, making the code more readable and maintainable.

1. Enum Benefits:
    - Improved code readability and maintainability.
    - Ensures that values are limited to a defined set
    - Provides a convenient way to iterate and compare values.

2. Use Cases:
    - Representing a fixed set of options or choices.
    - Mapping symbolic names to constant values.
    - Enhancing code clarity and preventing errors.

Enums

```python
##### enums

from enum import Enum

class Day(Enum):
    MONDAY = 1
    TUESDAY = 2
    WEDNESDAY = 3
    THURSDAY = 4
    FRIDAY = 5
    SATURDAY = 6
    SUNDAY = 7

print(Day)
print(Day.MONDAY)
print(Day.MONDAY.name)
print(Day.MONDAY.value)

print('\nobj != value')
print(Day.MONDAY==1)
print('\nvalue test')
print(Day.MONDAY.value==1)
print('\nis')
print(Day.MONDAY is Day.MONDAY)
# == delegated to is
print('\n==')
print(Day.MONDAY==Day.MONDAY)
print('\nin')
print(Day.MONDAY in Day)
```

Serialization

Serialization is the process of converting objects into a format suitable for storage or transmission. Pickle provides an easy and efficient way to serialize complex data structures, such as lists, dictionaries, and custom objects, into a binary representation.

Pickle Serialization

- Simplicity: pickle provides a simple API for serializing and deserializing objects.
- Full Object Serialization: pickle can handle complex data structures, including nested objects and custom classes.
- Efficient Storage: Serialized objects can be stored in files or transferred over networks.
- Data Integrity: Pickle ensures that objects are serialized and deserialized with integrity, preserving their internal structure.

Considerations and Limitations

- Security Risks: <u>untrusted pickle data can execute arbitrary code</u>, so be cautious when unpickling objects from untrusted sources.
- <u>Version Compatibility</u>: Pickle may have compatibility issues between different versions of Python or third-party libraries.
- <u>Non-Human Readable</u>: Serialized objects are stored in a binary format, making it difficult for humans to read and debug.

Exceptions

Synthax error = wrong python code -> "print())"
Exception = code is formally correct, but not semantically, at runtime -> a/0

Try Block: the code that may raise an exception is placed within a try block.
Except Block: the code in the except block is executed when an exception is raised.
Finally Block: the code in the finally block is executed regardless of whether an exception occurred or not.

Exception Handling Best Practices

- Be Specific: catch specific exceptions instead of using a generic except block.
- Graceful Recovery: provide appropriate error messages to the user.
- Logging: use logging to record information about exceptions for debugging purposes.
- Clean-Up Actions: use the finally block to release resources or perform clean-up actions.

Common Exception Types

ValueError: raised when a function receives an argument of the correct type but an inappropriate value.
TypeError: raised when an operation is performed on an object of an inappropriate type.
FileNotFoundError: raised when a file or directory is requested but cannot be found.
IndexError: raised when an index is out of range.
KeyError: raised when a key is not found in a dictionary.

Parallel programming

Python, *per se* is not ok for parallel programming:

- Huge difference between multi-threading and multi-processing.
- Python can do multi-threading but due to the GIL, they all scheduled to the same core.
- Multi-processing works, but many interpreters to be run (no shared memory), very clumsy unless you have trivially parallel long duration tasks.
- One can still use libraries as pytorch in ML, these are very fast or revert to compilers or PyCuda etc…

Decorators

Decorators are functions that wrap or modify other functions or classes. They provide a convenient way to enhance or extend the functionality of existing code.

Syntax: Decorators use the "@" symbol followed by the name of the decorator function. They are placed just before the function or class definition.

```
def decorator(F):
    print('called!')
    return F

@decorator
def myfunc():
    print('my function')

myfunc()
```

Decorators

One can also define class decorators

```
# class decorator
def decorator2(classVar):
    # global counter
    count = 0
    class DecoratorWrapper:
        # instance counter
        #count = 0
        def __init__(self,*args):
            print('just wrapped')
            self.wrapped = classVar(*args)
        # try to comment
        def __getattr__(self, name):
            nonlocal count
            #self.count += 1
            count += 1
            #print('retrieving count %d'%self.count)
            print('retrieving count %d'%count)
            return getattr(self.wrapped,name)
    return DecoratorWrapper

@decorator2
class myClass:
    ciao = 'ciao'
    def __init__(self):
        print('init')
```

Decorators

Nested decorators

```python
# nested decorators

def decA(F):
    print('A wrap')
    return F

def decB(F):
    print('B wrap')
    return F

@decB
@decA
def wrapMe():
    print('wrap me')

wrapMe()
```

Decorators

Decorators can have arguments

```
# decorators with arguments

def switcherDecorator(switchMe):
    def functionCatcher(F):
        def argsCatcher(*args,**kwargs):
            print('switch %d'%switchMe)
            if (switchMe):
                return args[0]()
            else:
                return F
        return argsCatcher
    return functionCatcher


@switcherDecorator(True)
def myStrangeFunction1(F=None):
    print('strange 1')

def myStrangeFunction2(F=None):
    print('strange 2')
```

Decorators

Decorators for methods need special care

```
def methodDecorator(decParam):
    def functionCatcher(F):

        def onCall(*args,**kwargs):
            print('catched on call')
            return F(*args,**kwargs)
        return onCall

        print('catched on call 2')
        return F
    return functionCatcher

class myClass():

    def __init__(self):
        print('init')

    @methodDecorator(True)
    def myMethod(self):
        print('myMethod')


mc = myClass()
mc.myMethod()
```

Decorators

Decorators can be used to implement design patterns as the singleton, or for a limited number of instances.

```
# decorator for the singleton pattern

def makeSingleton(myClass):
    instance = None
    def onCall(*args,**kwargs):
        nonlocal instance
        if (instance == None):
            instance = myClass()
        return instance
    return onCall


@makeSingleton
class myTargetClass():
    myName = 'Sergio Decherchi'
    def __init__(self):
        pass
```

Design Patterns

- Design patterns are reusable solutions to common problems that occur during software design and development.
- They provide proven approaches to solving specific design challenges and promote best practices in software development.
- Design patterns help improve code readability, maintainability, and extensibility, and they enable software engineers to communicate effectively about software design concepts.

The most used patterns:

- **Singleton**: ensures a class has only one instance, providing global access to it.
- **Decorator**: Dynamically adds new functionality to an object without altering its structure.
- **Object Iterator**: provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Factory Method**: defines an interface for creating objects, but lets subclasses decide which class to instantiate.
- **Adapter**: Converts the interface of a class into another interface that clients expect.

Numpy

Data Types

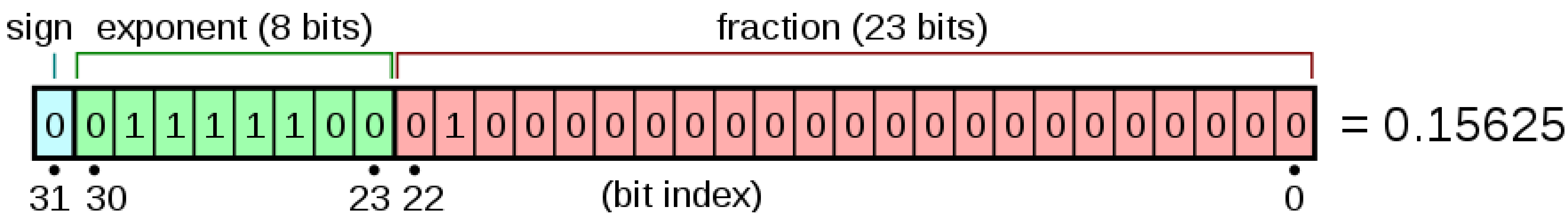
Table 4-2. NumPy data types

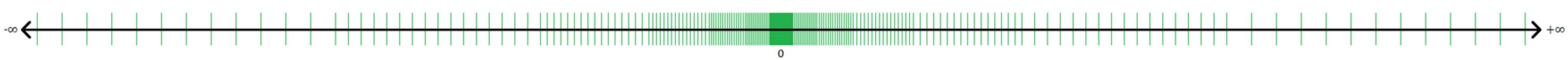
Type	Description
`int8, uint8`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	Signed and unsigned 16-bit integer types
`int32, uint32`	Signed and unsigned 32-bit integer types
`int64, uint64`	Signed and unsigned 32-bit integer types
`float16`	Half-precision floating point
`float32`	Standard single-precision floating point. Compatible with C float
`float64, float128`	Standard double-precision floating point. Compatible with C double and Python `float` object

Type	Description
`float128`	Extended-precision floating point
`complex64, complex128, complex256`	Complexnumbersrepresentedbytwo32,64,or128floats,respectively
`bool`	Boolean type storing `True` and `False` values
`object`	Python object type
`string_`	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use `'S10'`.
`unicode_`	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as `string_` (e.g. `'U10'`).

A word on floating point

$$(-1)^s \times 2^E \times M$$





Numpy, Scipy, Matplotlib

Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
Python does numerical computations slowly. 1000 x 1000 matrix multiply

Python triple loop takes > 10 min.
Numpy takes ~0.03 seconds

Numpy Features:

Typed multidimentional arrays (matrices)
Fast numerical computations (matrix math)
High-level math functions

Numpy-Arrays

```
##### numpy

import numpy as np

v0 = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
print(v0.ndim, v0.shape, v0.dtype)

v1 = np.arange(0,10,1)
print(v1)

v2 = np.linspace(0,10,11)
print(v2)

vzeros = np.zeros((5,2))
print(vzeros)

vones = np.ones((5,2))
print(vzeros)
```

Numpy-Broadcasting

```
vzeros = np.zeros((5,2))
vones  = np.ones((5,2))

sum1 = vzeros+vones
print(sum1)

sum2 = vones+1
print(sum2)

sum3 = vones+2*np.ones((1,2))
print(sum3)

sum4 = vones+4*np.ones((5,1))
print(sum4)
```

Numpy-Slicing

```
# vectors slicing

A = np.arange(20)
A = np.reshape(A,(5,4))

print('\n')
print(A[:,:])

print('\n')
print(A[0:1,:])

print('\n')
print(A[:,0:1])

# one can go out of bound
print('\n')
print(A[0:1000,0:1000])

print('\n\n')
skip = 2
print(A[0::skip,0::skip])
```

Numpy shaping

```
A = np.arange(20)
A = np.reshape(A,(5,4))

B = A.astype(np.int32)
C = A.astype(np.float32)

# unrolling
print('\n')
print(A.ravel())
# unrolling
# always copy
print('\n')
print(A.flatten())

A.transpose()
B = A.copy()
V1 = np.concatenate((A,A))
print('\nconcatenation 1')
print(V1)

V2 = np.concatenate((A,A),axis=1)
print('\nconcatenation 2')
print(V2)
```

Numpy stacking/dimensions

```
# stacking and dimensions

a = np.array([1, 2, 3])

b = np.array([4, 5, 6])

c = np.stack((a, b))
print(c)
c = np.stack((a, b),axis=-1)
print(c)

print(a.shape)
# all equivalent ways
a1 = a[..., np.newaxis]
print(a1.shape)

a1 = a[:, None]
print(a1.shape)

a1 = np.expand_dims(a, axis=-1)
print(a1.shape)
```

Numpy logical ops

```
# logical operations
a = np.array([1, 2, 3])
aorig = a.copy()
boolean = a>1.5
print(boolean.shape)
print(boolean.__class__)
print(a[boolean])

boolean2 = [False,True,True]
print(boolean2.__class__)
print(a[boolean])

# this is a view, does not copy
# hence it changes a
a[a>1.5]=0
print(a)
# this is a copy, if you change b
# it does change aorig
b = aorig[aorig>1.5]
print(b)
b[b>1.5]=0
print(b)
print(aorig)
```

Numpy saving/loading

```
# save and load

np.savez('data.npz',matrix=A,matrix2=V1)
data = np.load('data.npz')
AA = data['matrix']
print(AA)
VV1 = data['matrix2']
print(VV1)
```

Vector and Matrix Operations

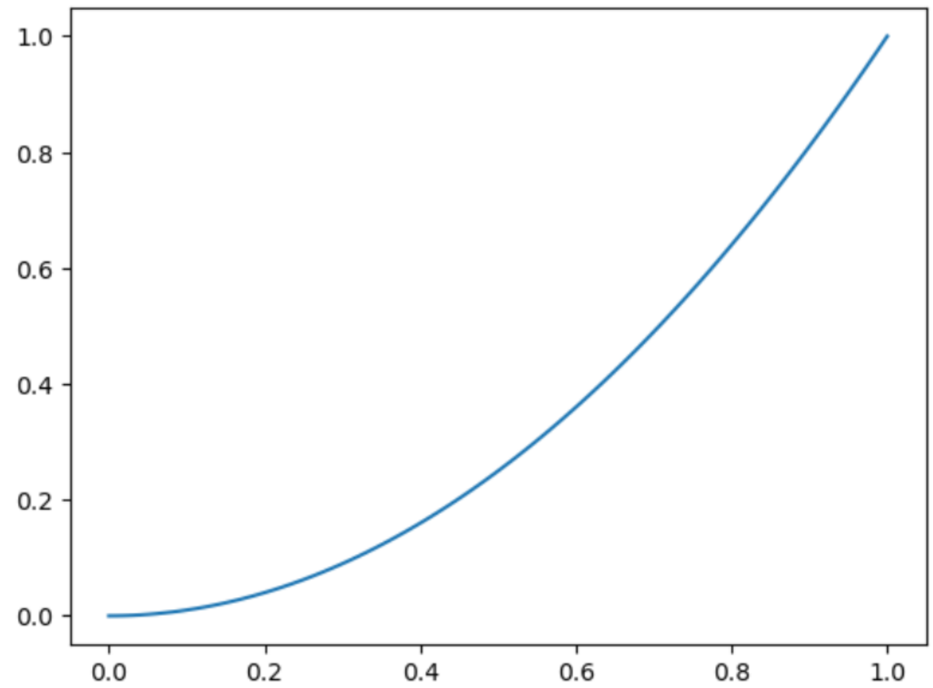
```
# dot, matrix mul and hadmard

A = np.arange(1,13)
print(A.shape)
dot = np.dot(A,A)
dot = np.dot(A,A.T)
print(dot)

# if you reshape transpose is relevant!
A = np.reshape(A,(12,1))
print(A.shape)
# this gives error
#dot = np.dot(A,A)
# this is ok
dot = np.dot(A,A.T)

A = np.reshape(A,(4,3))

# std matrix multiplication
B = np.matmul(A,A.T)
print(B)

# hadamard
C = np.multiply(A,A)
print(C)
```

Linear algebra

```
# load data
A = np.loadtxt('data.txt',dtype=np.float32)
print(A)

# linear system solution Ax=y
y = np.sum(A,axis=1)
print(y)

x = np.linalg.solve(A,y)
# check
print(x)
print(np.matmul(A,x))

eig = np.linalg.eigvals(A)
print(eig)

det = np.linalg.det(A)
print(det)

rank = np.linalg.matrix_rank(A)
print(rank)

U,S,V = np.linalg.svd(A)

print('U',U)
print('S',S)
print('V',V)
```

Matplotlib

```
import matplotlib.pyplot as plt

x = np.linspace(0,np.pi,100)
y = np.sin(x)

plt.plot(x,y)
plt.show()
```



Exercises

Install anaconda:

- Install anaconda
- Create an envinroment
- Install numpy and matplotlib

Sorting algorithm:

- Write a insertionSort algorithm: try writing a "Utils" class which wraps it as function. Load from file the input sequence and save the results on a file.

Numerical algorithm:

- Write a gradient descent algorithm for an arbitrary function 1d:
    - code the cost and gradient as functions; start from a simple function as a parabola y = a*x^2+b*x+c.
    - reads the parabola parameters from a file
    - check the when to stop the iterations
    - pass the cost function and its gradient to the gradient routine
    - allows a programmable learning rate
    - show the time evolution of the gradient via matplotlib (time.sleep()) to show frames in sequence.
    - study the effect of changing the value of the learning rate.

Consider managing exceptions, errors, files, classes and/or decorators. Can devise several versions?