Python

Adam Boult (www.bou.lt)

April 30, 2025

Contents

Preface

This is a live document, and is full of gaps, mistakes, typos etc.

Part I

Basic

Chapter 1

Numbers, booleans and dynamic typing in Python

1.1 Introduction

1.1.1 Introduction

ints and floats

complex

type checking. na checking

define as specific data type in python. long int etc

null/na etc in python. nan. inf

overflows of int etc size in python what happens if number gets too big?

1.1.2 Arithmetic

// in python is integer point division. / is floating point division

1.1.3 References and copying on write

```
a = 1000
b = a
```

This gives b the same address as a.

If we instead do the following then b will have a different reference after it is changed.

```
a = 1000
b = a
b = b + 1
```

1.1.4 Small integer caching

Everything in Python is an object.

Normally, when a number is referenced an object for it is created.

If the number is a small integer (between -5 and 256 inclusive) instead a reference to these objects are used.

1.1.5 Casting

1.1.6 Booleans

1.1.7 Dynamic typing and lack of generics

Don't need generics because of dynamic typing.

Uses ducktyping.

1.1.8 Type hints

```
x: int = 1
y: float = 2
```

Chapter 2

Lists, tuples and immutablility AND Lambda functions

2.1 Introduction

2.1.1 SORT

list slice:

`v[1:3]`

+ can also do third thing?

reverses???

`v[::-1]`

2.1.2 Introduction

Python get array size, size of other iterables, sets

map and filters run on iterable. also the loops? is a set an iterable?

+ conditional substitutions of lists in python

2.1.3 Lists

Python lists are dynamic arrays.

```
a = [1,2,3]
```

Lists are mutable.

How to do each of following:

concatenation

slice/filter

insert

pop

traverse

map

sort

reverse

2.1.4 List slices

2.1.5 List comprehensions

Eg the following defines a list, and then the second line returns that list.

```
my_list = [1,2,3]
[x for x in my_list]
```

can do functions too. following returns [3,4,5]

```
[x + 2 for x in my_list]
```

filter

```
[x for x in my_list if x >2]
```

can be on any iterable:

```
[x for x in range(10)]
```

2.1.6 Lambda functions

filter

+ map function + lambda functions

2.1.7 Tuples

```
a = (1,2,3)
```

Tuples are immutable.

why not just use copy on write instead of tuples? + complex to implement?

why tuples broadly safety + mean that if create one tuple based on another, deep copy? if mutable + a=(1,2,3) + b=a + b[0]=2 * this fails, but if it didn't we might get the following + print(a) * (2,2,3) + print(b) * (2,2,3)

with mutability can still do eg a=a[2] because this is creating a new thing and using it in name immutability means can't do eg b[3] = 2 allows optimiser to assume not mutable. can cause speed ups

Chapter 3

Control flow

3.1 Introduction

3.1.1 Introduction

generators as type of iterable

+ list(range()) to get actual enumerated

+ match/case in python

Enumerate Range

and or in python

+ iterate over list in python: enumerate + iterate over dict in python: for in dict.items() + iterate over copy to prevent problems in changing stuff. for x in dict.copy().items() + for i in range(5) + for i in range(0, 5, 1)

python: + question mark notation if else

logical functions in base: + any(); all()

3.1.2 enumerate

```
for count, value in enumerate(list_name):
```

3.1.3 zip

start with 2 (or more) lists or other things

want to pair them

```
for x in zip(a, b):
```

iterates over 1st of both, 2nd of both etc

Chapter 4

Strings

4.1 Introduction

4.1.1 Defining strings

```
x = "Hello"
```

Can type hint a string

```
x: str = "Hello"
```

4.1.2 String pool

Strings stored in a pool. If 100 variables are of the same string, not stored 100 times.

4.1.3 Immutability of strings

Can't do the following.

```
x = "Hello"
x[2] = "b"
```

For security (if not, presumably copy on write could be implemented).

4.1.4 Operations on strings

string strip

concatenation

4.1.5 Iterating over a string

Can treat strings like an array

Following prints out each character on a new row.

```
x = "Hello"
for ch in x:
    print(ch)
```

4.1.6 SORT

if define a string, is unicode. can convert to bytes

"my string".encode() to get the bytes which encode the string

b"b string".decode() to convert byte string into unicode string

eg for hebrew characters

```
"hellò".encode()
b'\xd7\xa9\xd7\x9c\xd7\x95\xd7\x9d \xd7\xa2\xd7\x95\xd7\x9c\xd7\x9d'.decode()
```

4.2 Printing strings

4.2.1 print

```
x = "apple"
print(x)
```

4.2.2 repl

```
x = "apple"
repl(x)
```

4.2.3 string.format()

4.2.4 F strings

```
x = "apples"
y = 2
print(f"I eat {y} {x}")
```

4.2.5 % operators

4.2.6 Using commas in print()

```
x = "apple"
y = banana"
print(a, b)
```

Part II

Functions

Chapter 5

Functions and more on dynamic typing

5.1 Introduction

5.1.1 SORT

passing arrays to functions. what is passed? reference ot first? length separately?

5.1.2 Defining functions

Best practice is to name them *lower_case_and_underscore*.

Everything including functions are objects. As a result functions are first class. Functions can accept functions and can return functions.

```
def my_function_no_parameters():
    return 0

def my_function_with_parameters(x, y):
    return x + y
```

5.1.3 Decorators

apply @ function to function below to decorate it. syntactic sugar

```
def my_decorator(func):
    def inner(a):
```

```
        print("Printing ", a, " in a decorated way")
        return
    return inner

@my_decorator
def just_printing(a):
    print(a)
```

5.1.4 Default parameters

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

5.1.5 Side effects of functions

Can have side effects on objects in parameters if mutable.

Can have side effects on global variables if present.

5.1.6 Passing lists and objects to functions rather than doing literally

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

Can accept multiple variables with *arguments

```
l=[1,2]
f(*l)
```

Or can accept named literals with **kwargs (ie key word arguments)

```
args = {"a":1, "b":2}
f(**args)
```

5.1.7 Main function in python

```
def main():
    // Do stuff
if __name__ == "__main__":
    main()
```

5.1.8 Generators and the yield function

functions which make generators: yield function

5.2 Documentation

5.2.1 Function annotations

```
def my_function(x: "annotation of the input variable x") -> "annotation of the return":
    return x
```

5.2.2 Type hints

Can but types in annotations. Types are not checked at run time.

```
def f(a: str, b: str = "apple") -> str:
    return a
```

5.2.3 Function documentation

defining functions: + triple quote comment at start for documentation

```
def my_function_no_parameters():
    """
    This is the documentation of my function.
    """
    return 0
```

5.3 Partial functions, Currying and lambda functions

5.3.1 Defining functions using lambdas

```
my_function = lambda a: a + 1
```

5.3.2 Closures in Python

5.3.3 Partial functions

```
from functools import partial

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

g = partial(f,
```

5.3.4 Currying

Part III

Classes

Chapter 6

Classes

6.1 Introduction

6.1.1 Instance methods and the init constructor

```
class MyClass:

    def __init__(self, name):
        self.name = name
        self.data = []

    def do_something(self, x):
        self.data.append(x)

myObject = MyClass("bob")
```

Or optionally with type hints:

```
myObject: MyClass = MyClass("bob")
```

6.1.2 Class documentation

```
class MyClass:

    """
    Documentation of class
    """
```

```
    def do_something(self, x):
        """Documentation of method"""
        self.data.append(x)
```

6.1.3 Global variables in classes and class methods

Need to use decorators for classmethod and staticmethod to get them to work properly, though possibly being phased out?

```
class MyClass:

    x = 1

    def __init__(self, name):
        self.name = name
        self.data = []

    @classmethod
    def do_something(cls, y):
        cls.x = y
```

6.1.4 Static methods

```
class MyClass:

    @staticmethod
    def do_something(x):
        print(x)
```

6.1.5 Getaddr and setaddr

Can be used to validate data, present properly and ensure encapsulation.

```
    __getaddr__


    __setaddr__
```

```
    __deladdr__
```

6.1.6 Class destructor

Called when object deleted (eg by garbage collector or "del myObject")

```
class MyClass:

    def __init__(self, name):
        self.name = name
        self.data = []

    def __del__(self):
        pass
```

6.1.7 Replacing built in functions

```
class MyClass:

    def __init__(self, things):
        self.things = things

    def __len__(self):
        return len(self.things)
```

6.1.8 Operator overloading

Can overload other operators too.

```
class MyClass:

    def __init__(self, val):
        self.val = val

    def __add__(self, other):
        return len(self.val + other.val)

x = MyClass(1)
y = MyClass(2)
x + y
```

6.1.9 Class iterators

```
__iter__
```

```
__next__
```

6.2 Inheritance

6.2.1 Inheritance

```
class BaseClass:

    x = 1

class DerivedClass(BaseClass):
    y = 1
```

6.2.2 Identifying inheritance

```
issubclass(class, classinfo)
```

6.2.3 Encapsulation in Python

Public: Accessible from anywhere Protected: Accessible from within class and subclass Private: Accessible from within class

Done with underscores. x below is public. y is protected. z is private.

```
class MyClass:

    def __init__(self, x, y, z):
        self.x = x
        self._y = y
        self.__z = z
```

Can also do these for methods

```
class MyClass:

    def a():
        print(1)
```

```
    def _b():
        print(2)
    def __c()
        print(3)
```

6.2.4 Getters and setters

Used for encapsulation and cleaning.

```
class MyClass:

    def __init__(self, x, y, z):
        self.x = x
        self._y = y
        self.__z = z

    def get_z(self):
        return(self.__z)
    def set_y(self, y):
        self._y = y
```

6.2.5 Multiple inheritance

```
class BaseClassA:

    x = 1
class BaseClassB:

    y = 1

class DerivedClass(BaseClassA, BaseClassB):
    z = 1
```

6.2.6 Super

Access methods and parameters from parent class.

```
class BaseClass:

    x = 1
```

```
class DerivedClass(BaseClassA):
    z = super().x + 1
```

6.2.7 Overwriting

```
class BaseClass:

    def do_thing(self):
        print(1)

class DerivedClass(BaseClassA):
    def do_thing(self):
        print(2)
```

6.2.8 Checking membership

Check is member of class or subclass.

```
isinstance(object, int)
```

Part IV

Error handling and memory management

Chapter 7

datetime

7.1 Introduction

7.1.1 try,except,else,finally

```
try:
    print(x)
except:
    print("exception occured")
```

can do different types of exception

```
try:
    c = a // b
except ZeroDivisionError:
    print("Dividing by zero")
except:
    print("Some other problem happened")
```

Can add else for if no exceptions

```
try:
    print(x)
except:
    print("Exception happend")
else:
    print("No exception happened")
```

Can add "finally" black to always execute regardly of type of exception or if it just did else

```
try:
    print(x)
```

```
except:
    print("Exception happened")
else:
    print("No exception happened")
finally:
    print("This prints if an exception happens or not")
```

7.1.2 Manually raising exceptions with "raise"

We can manually raise exceptions

```
raise Exception
raise ZeroDivisionError
```

7.1.3 assert

assert here too: assert(False) raises AssertionError

7.1.4 with

"with" function easier to use than try except released objects afterwards, so need to worry less about clean up if eg opening files

Chapter 8

Reference counting and del

8.1 Introduction

8.1.1 "del"

Second "print()" fails because we have released the variable.

```
x = 1000
print(x)
del x
print(x)
```

8.1.2 Reference counting in Python

Reference count on each object.

References tracked automatically.

If references hits 0, object is automatically deleted.

Chapter 9

Garbage collection and the heap in Python

9.1 Introduction

9.1.1 Reference counting in Python

9.1.2 Generational garbage collection in Python

Reference counting cannot detect circular references.

9.1.3 The heap

what it means to say y = x

x pointer to object in heap

y set to be pointer to same object

if change something with reference count above 1 "copy on write". confirm?

Part V

Cont.

Chapter 10

Unix shell integration

10.1 Introduction

10.1.1 Introduction

sys.argv to grab arguments

start with

`#!/usr/bin/env python3`

start with

```
    # -*- coding: utf-8 -*-
```

Part VI

Standard library

Chapter 11

Copy

11.1 Introduction

11.1.1 Introduction

Copy and deepcopy.

Chapter 12

os, sys, subprocess

12.1 Introduction

12.1.1 Introduction

12.1.2 os

12.1.3 sys

sys.argv. how to use?

```
python file_name.py my arguments
```

sys.argv[0] is eg "file_name.py"

sys.argv[1] is "my"

sys.argv[2] is "arguments"

12.1.4 eval

12.1.5 subprocess

Chapter 13

Reading and saving data with pickle, xml, csv

13.1 Introduction

13.1.1 Introduction

Chapter 14

math, cmath, random, statistics

14.1 Introduction

14.1.1 Introduction

Math package mostly a wrapper around C math.h.

Chapter 15

string and re

15.1 Introduction

15.1.1 Introduction

Compiling regexes.

Chapter 16

datetime

16.1 Introduction

16.1.1 Introduction

Chapter 17

doctest

17.1 Introduction

17.1.1 Introduction

Chapter 18

functools

18.1 Introduction

18.1.1 Introduction

"reduce" function

Partial functions

Chapter 19

dataclasses

19.1 Introduction

19.1.1 Introduction

Automatically creates boilerplate for classes which used to store data, including *__init__* function with the relevant parameters.

```
from dataclasses import dataclass

@dataclass
class MyClass:

    x: str
    y: float
    z: int = 0
```

generates among other methods:

```
class MyClass:

    def __init__(self, x: str, y: float, z: int = 0):
        self.x = x
        self.y = y
        self.z = z
```

Chapter 20

Logging

20.1 Introduction

20.1.1 Introduction

Chapter 21

gc

21.1 Introduction

21.1.1 Introduction

changing garbage collector behaviour

```
import gc
gc.disable()
```

Chapter 22

dis

22.1 Introduction

22.1.1 Introduction

Disassembly of python code to see opcodes etc for virtual machine.

Part VII

Standard library: Data types

Chapter 23

Linked lists, sets, queues, stacks and nodes

23.1 Introduction

23.1.1 Array

Array not in default data types. List is different: Dynamic, can hold different data types.

import array

how to do each of the following:

concatenation

slice/filter

insert

pop

traverse

map

sort

reverse

23.1.2 Nodes

```
class Node(object):
    def __init__(self):
```

```
        self.data = None
        self.next = None
```

23.1.3 Linked Lists

```
class LinkedList:
    def __init__(self):
        self.current_node = None

    def add_node(self, data):
        new_node = Node()
        new_node.data = data
        new_node.next = self.current_node
        self.current_node = new_node

    def print(self):
        node = self.current_node
        while node:
            print node.data
            node = node.next

linkedList = LinkedList()
linkedList.add_node(1)
linkedList.add_node(2)
linkedList.add_node(3)
linkedList.add_node(4)

ll.print()
```

concatenation

slice/filter

insert

pop

traverse

map

sort

reverse

23.1.4 Sets

Uses a hash table.

Following returns $1, 3, 5$.

```
a = {1,3,5,3}
a
```

23.1.5 Queues

Can use a list, but is slow. Requires moving everything around.

Short for double ended queue.

collections.deque

also can do queue.Queue but seems to be based on collections.deque

Double ended queues are implemented as double linked lists.

23.1.6 Stacks

Can also use a list, but can be slower if expanding array?

Like for queue, can use collections.deque

Part VIII

Standard library: Parallel processing and async

Chapter 24

Multithreading and the Global Interpreter Lock (GIL)

24.1 Introduction

24.1.1 Introduction

Chapter 25

threading

25.1 Introduction

25.1.1 Introduction

which packages do each of these 3 use?

25.1.2 asyncio

subject to gil?

25.1.3 threaded

subject to gil? only 1 thread at a time. other thread like io mean gil go to another thread, so still faster. ie, only one thread running at a time, but if one thread is waiting for io response, can go to other thread.

25.1.4 multiprocessing

(pool, process). in pool, each process has own gil. map, imap, imap unordered. used to get around gil by having multiple concurrent threads.

Part IX

Implementations of Python

Chapter 26

cpython and pypy

26.1 Introduction

26.1.1 Introduction

python: + cpython (not jit?) + pypy (jit?)

why jit? can't do aot if dynamic types, array sizes. jit vs interpreter allows speed up

default is cpython pypy is alternative

cpython: can compile to .pyc bytecode which it then interprets

Part X

Writing modules and libraries

Chapter 27

Writing modules using python

27.1 Introduction

27.1.1 Introduction

Packaging and installing code.

Chapter 28

Writing modules using c and cython

28.1 Introduction

28.1.1 Introduction

writing modules in c

```
#include <Python.h>
```

in c code

Part XI

Accessing modules

Chapter 29

pip

29.1 Introduction

29.1.1 Introduction

```
python -m pip install pandas
python -m pip install --user pandas
python -m pip install --upgrade pandas
python -m pip install git+<path_to_git>
```

29.1.2 requirements.txt

pip install -r requirements

can have pip install git+https equiv in there

python -m pip freeze ¿ requirements.txt

can make manually instead. better because freeze puts dependencies too

29.1.3 Pipfile

Pipfile is replacement for requirements.txt

python -m pip install -p ¡file¿

Pipfile generates Pipfile.lock can run "pipenv lock" pipfile listed python version expected

29.1.4 Mirroring pip

can set up mirror and use pip config to access it

Part XII

Managing different package versions

Chapter 30

venv

30.1 Introduction

30.1.1 Introduction

venv is in python standard library

```
python -m venv path/to/venv
source path/to/venv/bin/activate
```

30.1.2 virtualenv

virtualenv is not in standard library. Alternative to venv, don't need to use.

30.1.3 virtualenvwrapper

virtualenvwrapper is extensions for virtualenv

Part XIII

Managing different Python versions

Chapter 31

Managing multiple versions of Python with pyenv and pipenv

31.1 Introduction

31.1.1 Introduction

pyenv (and pyenv-virtualenv and pyenv-virtualenvwrapper)

pyenv-virtualenv allows use of virtualenv and pyenv

eg can run

```
pyenv install 2.7.15
```

```
pyenv uninstall 2.7.15
```

to list installed versions

```
pyenv versions
```

switch to version:

```
pyenv global 2.7.15
```

```
pyenv local 2.7.15
```

```
pyenv shell 2.7.15
```

restore:

```
pyenv global system
```

now python/pip will use the python version in question

31.1.2 pipenv

basically integrates adn replaces pip and venv

uses Pipfile

automatically creates virtual environments

```
pipenv install pandas
```

automatically works with Pipfile and creates if needed

```
pipenv uninstall pandas
```

```
pipenv run python main.py
```

update lock file

```
pipenv lock
```

install from lock

```
pipenv sync
```

does lock and sync

```
pipenv update
```

spawns shell in environment. exit with exit()

```
pipenv shell
```

see dependency graph

```
pipenv graph
```

Part XIV

Interactive Python interpreters

Chapter 32

The Python interpreter

32.1 Introduction

32.1.1 Introduction

Python help function help(print)

Chapter 33

Jupyter

33.1 Introduction

33.1.1 Introduction

use of % at start of line in jupyter(ipython). tells interpreter how to behave?

use of ! at start of line means run in shell.

Part XV

conda

Chapter 34

conda

34.1 Introduction

34.1.1 Introduction and installing miniconda

miniconda allows the creation of Python environments, including Python versions. Pip can be used inside these environments, and in addition other non-Python packages can be installed inside these environment.

Miniconda is not available on official repos.

34.1.2 The "base" environment and creating conda environments

list environments

```
conda info --envs
```

environment.yml conda env create -f environment.yml + creates from environment file

conda install packages conda virtual environemnts

```
conda create --name <env_name>
conda create -n myenv python=3.9
conda install -n myenv scipy
conda install -n myenv scipy=0.17.3
conda create -n myenv python=3.9 scipy=0.17.3 astroid babel
conda create --prefix ./envs jupyterlab=3.2 matplotlib=3.5 numy=1.21
conda activate ./envs
conda deactivate
```

34.1.3 Installing packages in conda environments

pip in conda

if we install pip when making environment, can install using pip inside conda within environemtns

```
conda list (list packages)
conda install scipy
```

update environment

```
conda env update --prefix ./env --file environment.yml --prune
```

34.1.4 Deleting conda environments

34.1.5 .condarc

.condarc (has default packages for new environemnts?)

The condarc file contains info on which environment should /.condarc

34.1.6 Setting up a mirror for conda

page on local mirror using conda-mirror

34.1.7 anaconda

anaconda is miniconda but with additional default packages.