# Python

Adam Boult (www.bou.lt)

February 10, 2023

# 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 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

### 2.1.2 Lists

Python lists are dynamic arrays.

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

Lists are mutable.

### 2.1.3 List slices

### 2.1.4 List comprehensions

### 2.1.5 Lambda functions

filter

+ map function + lambda functions

### 2.1.6 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

# Reference counting and del

## 3.1 Introduction

### 3.1.1 "del"

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

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

### 3.1.2 Reference counting in Python

Reference count on each object.

References tracked automatically.

If references hits 0, object is automatically deleted.

# Chapter 4

# Control flow

## 4.1 Introduction

### 4.1.1 Introduction

generators as type of iterable

+ list(range()) to get actual enumerated

+ zip function

+ match/case in python

Enumerate Range Zip

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()

# Chapter 5

# Strings

## 5.1 Introduction

### 5.1.1 Defining strings

```
x = "Hello"
```

Can type hint a string

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

### 5.1.2 String pool

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

### 5.1.3 Immutability of strings

Can't do the following.

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

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

### 5.1.4 Operations on strings

string strip

concatenation

### 5.1.5 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)
```

## 5.2 Printing strings

### 5.2.1 print

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

### 5.2.2 repl

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

### 5.2.3 string.format()

### 5.2.4 F strings

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

### 5.2.5 % operators

### 5.2.6 Using commas in print()

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

# Chapter 6

# Functions and more on dynamic typing

## 6.1 Introduction

### 6.1.1 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
```

### 6.1.2 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)
```

### 6.1.3 Default parameters

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

### 6.1.4 Side effects of functions

Can have side effects on objects in parameters if mutable.

Can have side effects on global variables if present.

### 6.1.5 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)
```

### 6.1.6 Main function in python

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

### 6.1.7 Generators and the yield function

functions which make generators: yield function

## 6.2 Documentation

### 6.2.1 Function annotations

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

### 6.2.2 Type hints

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

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

### 6.2.3 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
```

## 6.3 Partial functions, Currying and lambda functions

### 6.3.1 Defining functions using lambdas

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

### 6.3.2 Closures in Python

### 6.3.3 Partial functions

```
from functools import partial

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

g = partial(f,
```

### 6.3.4 Currying

# Chapter 7

# Classes

## 7.1 Introduction

### 7.1.1 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")
```

### 7.1.2 Class documentation

```
class MyClass:

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

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

### 7.1.3 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
```

### 7.1.4 Static methods

```
class MyClass:

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

### 7.1.5 Getaddr and setaddr

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

```
    __getaddr__


    __setaddr__
```

```
    __deladdr__
```

### 7.1.6 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
```

### 7.1.7 Replacing built in functions

```
class MyClass:

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

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

### 7.1.8 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
```

### 7.1.9 Class iterators

```
__iter__
```

```
__next__
```

## 7.2 Inheritance

### 7.2.1 Inheritance

```
class BaseClass:

    x = 1

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

### 7.2.2 Identifying inheritance

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

### 7.2.3 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)
```

### 7.2.4 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
```

### 7.2.5 Multiple inheritance

```
class BaseClassA:

    x = 1
class BaseClassB:

    y = 1

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

### 7.2.6 Super

Access methods and parameters from parent class.

```
class BaseClass:

    x = 1
```

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

### 7.2.7 Overwriting

```
class BaseClass:

    def do_thing(self):
        print(1)

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

### 7.2.8 Checking membership

Check is member of class or subclass.

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

# Chapter 8

# Garbage collection

## 8.1 Introduction

### 8.1.1 Reference counting in Python

### 8.1.2 Generational garbage collection in Python

Reference counting cannot detect circular references.

# Part II

# Cont.

# Chapter 9

# Unix shell integration

## 9.1 Introduction

### 9.1.1 Introduction

sys.argv to grab arguments

start with

`#!/usr/bin/env python3`

start with

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

# Chapter 10

# The Python interpreter

## 10.1 Introduction

### 10.1.1 Introduction

Python help function help(print)

# Chapter 11

# Multithreading and the Global Interpreter Lock (GIL)

## 11.1 Introduction

### 11.1.1 Introduction

# Chapter 12

# Debugging

## 12.1 Introduction

### 12.1.1 Introduction

Logging

stop script from anywhere without throwing

print to log

assert try catch exception handling raise

# Part III

# Standard library

# Chapter 13

# Copy

## 13.1 Introduction

### 13.1.1 Introduction

Copy and deepcopy.

# Chapter 14

# os, sys, subprocess

## 14.1 Introduction

### 14.1.1 Introduction

os.system/eval/subprocess

# Chapter 15

# Reading and saving data with pickle, xml, csv

## 15.1 Introduction

### 15.1.1 Introduction

# Chapter 16

# math, cmath, random, statistics

## 16.1 Introduction

### 16.1.1 Introduction

Math package mostly a wrapper around C math.h.

# Chapter 17

# string and re

## 17.1 Introduction

### 17.1.1 Introduction

Compiling regexes.

# Chapter 18

# doctest

## 18.1 Introduction

### 18.1.1 Introduction

# Chapter 19

# functools

## 19.1 Introduction

### 19.1.1 Introduction

"reduce" function

# Chapter 20

# dataclasses

## 20.1 Introduction

### 20.1.1 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 21

# dis

## 21.1 Introduction

### 21.1.1 Introduction

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

# Part IV

# Standard library: Data types

# Chapter 22

# linkedListsSetsQueueStackNodes

## 22.1 Introduction

### 22.1.1 Array

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

import array

### 22.1.2 Nodes

```
class Node(object):
    def __init__(self):
        self.data = None
        self.next = None
```

### 22.1.3 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()
```

### 22.1.4 Sets

Uses a hash table.

Following returns $1, 3, 5$.

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

### 22.1.5 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.

### 22.1.6 Stacks

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

Like for queue, can use collections.deque

# Part V

# Standard library: Parallel processing and async

# Chapter 23

# threading

## 23.1 Introduction

### 23.1.1 Introduction

which packages do each of these 3 use?

### 23.1.2 asyncio

subject to gil?

### 23.1.3 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.

### 23.1.4 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 VI

# Implementations

# Chapter 24

# cpython and pypy

## 24.1 Introduction

### 24.1.1 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 VII

# Writing modules and libraries

# Chapter 25

# Writing modules using python

## 25.1 Introduction

### 25.1.1 Introduction

Packaging and installing code.

# Chapter 26

# Writing modules using c and cython

## 26.1 Introduction

### 26.1.1 Introduction

writing modules in c

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

in c code

# Chapter 27

# pip and venv

## 27.1 Introduction

### 27.1.1 Introduction

page on pip + pip install -r requirements

python venv

# Part VIII

# Basic scientific libraries

# Chapter 28

# NumPy

## 28.1 Introduction

### 28.1.1 Introduction

get numpy size,

numpy algebra: + *,@,+,- + np.load, np.save

numpy + data structures

# Chapter 29

# Matplotlib

## 29.1 Introduction

### 29.1.1 Introduction

designed around numpy but can use math module

# Chapter 30

# SciPy

## 30.1 Statistics

### 30.1.1 Introduction

SciPy includes statistics functionality.

Built on numpy and matplotlib.

# Part IX

# Other scientific libraries

# Chapter 31

# pandas

## 31.1 Introduction

### 31.1.1 Introduction

built on numpy.

get pandas size

sort pandas

pandas: subsetting tables: .loc[] + selecting series from df + selecting row from df selecting element from series/row

pandas: apply

missing value, joins

pandas data structures

# Chapter 32

# seaborn

## 32.1 Introduction

### 32.1.1 Introduction

Works better with pandas natively than matplotlib?

Abstraction around matplotlib.

# Chapter 33

# sklearn

## 33.1 Introduction

### 33.1.1 Introduction

uses numpy, matplotlib/plotly, pandas, scipy

# Chapter 34

# biopython

## 34.1 Introduction

### 34.1.1 Introduction

Builds on numpy.

# Chapter 35

# plotly

## 35.1 Introduction

### 35.1.1 Introduction

# Chapter 36

# statsmodels

## 36.1 Introduction

### 36.1.1 Introduction

builds on numpy, pandas, matplotlib

# Part X

# Tensor libraries

# Chapter 37

# tensorflow and keras

## 37.1 Introduction

### 37.1.1 Introduction

keras is "front end" for tensorflow

XLA (Accelerated linear algebra)

Gradient accumulation?

# Chapter 38

# pytorch

## 38.1 Introduction

### 38.1.1 Introduction

making a tensor: torch.tensor

`torch.from_numpy`
`torch.ones_like`
`torch.ones`
`torch.rand`
`torch.zeroes`

diff from numpy 1. device type (can attach cuda cores) 2. requires_grad (property of derivative)

linear algebra on tensors: * *@-+ * diff in any from numpy

thing on tensors: * nn.Linear * nn.ReLU * nn.Softmax

other tensor stuff * torch.save/torch.load * diff inf any from numpy

data: * torch.utils.data.DataLoader * torch.utils.data.Dataset

networks * nn.Module * nn.sequential * feedforward * backprob 1. torch autograd

specific types of models * torchvision (images) 1. datasets available * torchtext 1. datasets available * torchaudio 1. datasets available

JIT compiling and pytorch

XLA and pytorch * XLA is on TPU?

fast.ai is a thing that sits on top of pytorch

# Chapter 39

# jax

## 39.1 Introduction

### 39.1.1 Introduction

# Part XI

# Other

# Chapter 40

# Jupyter

## 40.1 Introduction

### 40.1.1 Introduction