# Contents

# Chapter 1

# Asynchronous Programming

- a style of concurrent programming
- doing many things at once

## How does python do multiple things at once

### Multiple Processes

- The OS does all the multi-tasking work
- Only option for multi-core concurrency

### Multiple Threads

- The OS does all the multi-tasking work
- In CPython, the GIL(Global Interpreter Lock) prevents multi-core concurrency

## Asynchronous Programming

- No OS intervention
- One Process, one thread

A style of concurrent programming in which tasks release the CPU during *waiting periods*, so that other *tasks* can use it.

# Chapter 2

# How is async is implemented

- Async functions need the ability to suspend and resume
- A function that enters a waiting period is suspended, and only resumed when the wait is over
- Four ways to implement suspend/resume in Python
  - Call back function
  - Generator functions
  - Async/await (Python 3.5)
  - Greenlets(require greenlet packages)
- Never pass mutable datatypes to a functions as arguments
- Instance variables and class variables
- static, public and protected variables in python
- decorators # annotations ??
- classmethods
  - (**???**)
  - class methods as alternative constructors
  - take cls as default argument, so mandatory for usage.
  - different from static methods
- static methods
  - (**???**)
  - static methods don't take cls or instance arguments as default argument
- Method resolution order
- Every class inherits from `builtins.object` like `Object()` class in java

## Inheriting from super class

- `super().__init__(first, last, pay)`
- `Employee.__init__(self, first, last, pay)`
- `isinstance()` and `issubclass()`
- `threading.Thread.__init__(self)`

## First class functions

```
def yell(text):
    return text.upper() + '!'
```

```
>>> yell('hello')
'HELLO!'
```

all data in python programs is represented by objects or relations between objects

```
>>> bark = yell
>>> bark('woof')
'WOOF!'
```

- function objects and their names are two seperate concerns. we can delete the function's original name `yell`
- But another name `bark` still points to the underlying function

```
>>> del yell
>>> yell('hello?')
# will give a NameError

>>> bark('hey')
'HEY!'
```

## Chapter 3

# loop like a native: loops in python

### iterable produces a stream of values.

### Assign stream values to name

### Execute statements once for each value in iterable

### Iterable decides what values it produces

### Lots of different things are iterable

**sum(iterable)**

**min(iterable)**

**max(iterable)**

```
for line_num, line in enumerate(f, start=1):
    # iterable only through enumerate
    # cannot be iterated using range(len(list))
```

# Chapter 4

# Meta class

```
from peewee import *

db = SqlliteDatabase('students.db')

class Student(Model):
    username = CharField(max_length=255, unique=True)
    points = IntegerField(default=0)

    class Meta:
    database = db
```

Think of `Meta` class as a container for configuration attributes of the outer class. The attributes of a class (for those that inherit the `Model`) are expect to be fields that corresponds to their counterparts in the database.

How then to add attributes that aren't database fields? The `Meta` class is the container for these non-field attributes.

## Classes are objects too

In python, everything is an object. and that includes classes. In python classes are `first class objects`

- they can be created at runtime, passed as parameters and returned from functions, and assigned to variables.

  def make_myklass(**kwattrs): return type('MyKlass', (object, ), dict(**kwattrs))

  my_klass_foo_bar = make_myklass(foo=2, bar=4) print(my_klass_foo_bar)

  x = my_klass_foo_bar() print(x.foo, x.bar)

Here we use the 3-argument form of the type built-in function to dynamically create a `class` name `MyKlass` inheriting from `object` with some attributes provided as arguments. Then we create one such class.

`my_klass_foo_bar` is equivalent to:

```
class MyKlass(object):
    foo = 2
    bar = 4
```

But it was created at runtime, returned from a function and assigned to a variable.

```
>>> class SomeKlass(object): pass
...
>>> someobject = SomeKlass()
>>> someobject.__class__
<class '__main__.SomeKlass'>
>>> SomeKlass.__class__
<class 'type'>
>>>
```

we've created a class and an object of that class. Examining the `__class__` of `someobject` we saw that it's `SomeKlass`. Next comes the interesting part. What is the class of `SomeKlass`? we see it's `type`

So `type` is the class of python classes. In other words, while in the example above `someobject` is a `SomeKlass` object, `SomeKlass` is itself a `type` object. Thus built-in class (type) is serving the role of being the class of classes.

## Meta class

A metaclass is defined as "the class of a class". Any class whose instances are themselves classes, is a `metaclass`. So, according to what we've seen above, this makes `type` a `metaclass` too - which is the most commonly used `metaclass` in python.

The purpose of `metaclass` is not to replace the class/object distinction with metaclass/class - it is to change the behaviour of class definitions (and thus their instances) in some way.

# Chapter 5

# Data visualization

- matplotlib
- pandas
- seaborn

## jupyter notebooks

- bokeh
- plotly
- D3.js

https://stackoverflow.com/a/38317060/5766752

# Chapter 6

# Table of Contents

# Chapter 7

# 7 steps to machine learning

machine learning is finding data to give answers

## gathering of data

- color
- alcohol

color (nm)

Alcohol %

Beer or Wine?

## Data Preparation

- put the data
- randomize the data
- Training and Evaluation
  - adjusting and correction of data preparation

## Choosing a model

- image based data
- text based
- music based

## Training

- weights
- biases
- prediction –> Test and update

## Evaluation data

- Training 80%
- Evaluation 20%

## Hyper parameter Tuning

- adjusting the hyper parameters

## Prediction

# Chapter 8

# Wrangling data with Pandas

manipulate the data for data science

## panel data –> Pandas

## dataframe

DataFrame.head() DataFrame.describe()

## Shuffling the data

## DataFrame access

- Columns –> use bracket notation
- Rows –> iloc[]
    - Range access

# Chapter 9

# Pygame introduction

- The main steps in a pygame application are :
  1. importing the pygame library
  2. Initialising the pygame library
  3. Creating a window
  4. Initialise game objects
  5. Start the game loop
- `pygame.display.flip()` update the screen every time

function's `__name__` won't affect how you can access it from your code. This identifier is merely a debugging aid. A *variable pointing* to a *function* and the *function* itself are two seperate concerns

there's also `__qualname__` which serves a similar purpose and provides a qualified name string to disambiguate function and class names

# Chapter 10

# Functions can be stored in data structures

## Chapter 11

# Functions can be passed to other functions

# Chapter 12

# Functions can be nested

## Chapter 13

# Higher order functions

- Functions that can accept other functionss as arguments are also called *higher-order functions*. They are a necessity for the functional programming style.
- The classical example for higher-order functions in python is the built-in `map` function.
- it takes a function and an iterable and calls the function on each element in the iterable yielding the results as it goes along.

```
def yell(text):
    return text.upper() + '!'

>>> list(map(yell, ['hello', 'hey', 'hi']))
['HELLO!', 'HEY!', 'HI!']
```

- map has gone through the entire list and applied the `yell` function to each element

# Chapter 14

# python's super

- dependency injection
- dependency injection using super
- super is introduced for one particular reason i.e., co-operative multiple inheritance
- python's super is not the same as other languages super
- python's super calls the children's ancestors
- MRO – method resolution order
- linearization is the solution for multiple inheritance

consider the example:

```
class Adam(object): pass
class Eve(object): pass
class Ramon(Adam, Eve): pass
class Gayle(Adam, Eve): pass
class Raymond(Ramon, Gayle): pass
class Dennis(Adam, Eve): pass
class Sharon(Adam, Eve): pass
class Rachel(Dennis, sharon): pass
class Matthew(Raymond, Rachel): pass
```

- The MRO will be in the form:

```
class Matthew(Raymond, Rachel):
    Method Resolution Order
        Matthew
        Raymond
        Ramon
        Gayle
        Rachel
        Dennis
        Sharon
        Adam
        Eve
        builtins.object
```

- super means the next one in the line.

- doesn’t mean the parents.
- children come before parents and parents stay in order

# Chapter 15

# Synchronization in Python

## synchronization primitives

- synchronization primitives are used to tell the program to keep the threads in synchrony.
- blocking methods are the methods which block execution of a particular thread until some condition is met.

## Locks

- A `Lock` has only two states:
  - locked
  - unlocked
- It is created in the unlocked state and has two principal methods.
  - `acquire()`
  - `release()`

The `acquire()` method locks the `Lock` and blocks the execution until the `release()` method in some other coroutine sets it to unlocked.

Then it locks the `Lock` again and returns `True` The `release()` method should only be called in locked state, it sets the state to unlocked and returns immediately. If `release()` is called in unlocked state, a `RunTimeError` is raised.