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

Data visualization

- matplotlib
- pandas
- seaborn

jupyter notebooks

- bokeh
- plotly
- D3.js

Chapter 4

Exceptions in python

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.

When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

For example, if function `A` calls function `B` which in turn calls function `C` and an exception occurs in function `C`. If it is not handled in `C`, the exception passes to `B` and then to `A`.

If never handled, an error message is thrown out and our program comes to a sudden unexpected fault.

```
try:
    # do something
    pass
except ValueError:
    # handle ValueError exception
    pass
except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass
except:
    # handle all other exceptions
    pass
```

- If an exception occurs during the execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after `try` statement.

Raising Exceptions

In Python Programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using keyword raise.

```
try:
    a = int(input("Enter positive integer: "))
    if a <= 0:
        raise ValueError("Not a positive number.")
```

```
except ValueError as ve:
    print(ve)
```

try … finally

The try statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources.

for example, we may be connected to a remote data center through the network or working with a file or with a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resources once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

```
try:
    f = open("test.txt", encoding='utf-8')
    # perform file operations
finally:
    f.close()
```

finally makes a difference if we return early

```
try:
    run_code_one()
except TypeError:
    run_code_two()
    return None
finally:
    other_code()
```

- finally block is run before the method returns in the above construct compared to :

```
try:
    run_code_one()
except TypeError:
    run_code_two()
    return None

other_code()
```

- the `other_code()` doesn't run if there's an exception.

other situations that can cause differences:

- If an exception is thrown inside the except block.
- If an exception is thrown in `run_code_one()` but it's not a `TypeError`.
- Other control flow statements such as `continue` and `break` statements.

Custom Exceptions

we may need to create custom exceptions that serves your purpose.

In python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class. Most of the built-in exceptions are also derived from this class.

```
class CustomError(Exception):
    pass

raise CustomError

raise CustomError("An Error occured")
```

In development, It is a good practice to place all the user-defined exceptions that our program raises in a seperate file. Many standard modules do this. They define their exceptions seperately as `exceptions.py` or `errors.py`.

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around – an except clause listing a derived class is not compatible with a base class). For example consider the following code.

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

the following code will print B, C, D in that order.

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B – the first matching except clause is triggered.

Chapter 5

Generators

- consider a function to square every number in a list and gives the result in the form of a list.

```
def square_numbers(nums):
    result = []
    for i in nums:
        result.append(i * i)
    return result

my_nums = square_numbers([1, 2, 3, 4, 5])

print my_nums
```

- we can convert the function `square_numbers` into generator. "'python def square_numbers(nums): for i in nums: yield (i * i)

my_nums = square_numbers([1, 2, 3, 4, 5])

for num in my_nums: print(num) "'

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

```
from threading import Thread
import time


def timer(name, delay, repeat):
    print("Timer: " + name + " Started")
    while repeat > 0:
        time.sleep(delay)
        print(name + ": " + str(time.ctime(time.time())))
        repeat -= 1

    print("Timer: " + name + " Completed")


def main():
    t1 = Thread(target=timer, args=("timer1", 1, 5))
    t2 = Thread(target=timer, args=("timer2", 2, 5))
    t1.start()
    t2.start()
    print("main completed")


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

Chapter 6

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 7

Functions can be stored in data structures

Chapter 8

Functions can be passed to other functions

Chapter 9

Functions can be nested

Chapter 10

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
- topological sorting of the dependency graph

Chapter 11

Bottom Up approach

- if we write the program bottom up, the function layout will match the logic flow - it'll go from the fully independent building blocks to the ones that depend on their results.

Chapter 12

top down approach

- for a top-down approach you'd flip the same structure on its head and start with the highest-level building block first, fleshing out the details later.
- vars – function
- key sharing dictionaries
- ordered dicts are not only ordered but also very less size compared to 2.7 dicts
- seperate chaining
- pprint – very helpful for nested datastructures.

Chapter 13

pythonNotes

python notes on important and tricky concepts

Chapter 14

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.