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
  - @classmethod
  - class methods as alternative constructors
  - take cls as default argument, so mandatory for usage.
  - different from static methods
- static methods
  - @staticmethod
  - 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

Decorators

Functions in Python are first-class

Unlike other languages, python treats functions as first-class citizens. That means that the language treats functions and data in the same way. Like data, functions can be assigned to variables, copied, and used as return values. They can also be passed into other functions as parametets.

```
def foo():
    print('foo!')

def bar(function):
    function()

bar(foo)
# prints 'foo!'
```

Decorators wrap existing functions

A decorator is a callable that takes a callable as an argument and returns another callable to replace it.

The @ syntax

The `@` syntax, is used to specify that the function `bar` should be wrapped or *decorated* by **foo**. The following statement is exactly equivalent to using `@`.

```
def foo(fn):
    def inner():
        print('About to call function.')
        fn()
        print('Finished calling function.')
    return inner

@foo
def bar():
    print('Calling function bar.')
```

```
bar = foo(bar)
```

Chapter 5

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 6

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.

```
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)
```

Chapter 7

Python look up loop holes

Late binding closures

Another common source of confusion is the way Python binds its variables (or in the surrounding global scope)

```
def create_multipliers():
    return [lambda x: i * x for i in range(5)]

for multiplier in create_multipliers():
    print(multiplier(2), end="..")
print()
```

output:

```
# not 0..2..4..6..8
8..8..8..8..8..
```

Five functions are created; instead all of them just multiply x by 4. Why? Python's closures are *late* binding. This means that the values of variables used in closures are looked up at the time the inner function is called.

Here, whenever any of the returned functions are called, the value of `i` is looked up in the surrounding scope at call time. By then, the loop has completed, and `i` is left with its final value of 4

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 8

Pipes in Python

Small elements are put together by their standard streams, i.e., the output of one process is used as the input of another process. To chain processes like this, so-called anonymous pipes are used.

Generally there are two kinds of pipes: * anonymous pipes and * named pipes

Anonymous pipes exist solely within processes and are usually in combination with forks.

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
- topological sorting of the dependency graph

Chapter 14

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 15

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 16

Global, Local and non local variables

Variables in python are implicitly declared by defining them.

The way python uses global and local variables is maverick. While in many or most other programming languages variables are treated as global if not otherwise declared, Python deals with variables the other way around. They are local, if not otherwise declared. The driving reason behind this approach is that global variables are generally a **bad practice** and should be avoided.

In most cases where we are tempted to use a global variable, it is better to utilize a parameter for getting a value into a function or return a value to get it out.

So when we define variables inside a function definition, they are local to this function by default. This means that anything you will do to such a variable in the body of the function will have no effect on other variables outside of the function, even if they have the same name.

This means that the function body is the scope of the block, where they are declared and defined in. They can only be used after the point of their declaration.

Non local variables

python3 introduces non local variables as a new kind of variables. Non local variables have a lot in common with global variables.

One difference to global variables lies in the fact that it is not possible to change variables from the module scope i.e. variables which are not defined inside of a function, by using the nonlocal statement.

for example:

```
def f():
    global x
    print(x)

x = 3
f()
```

The program will return number 3 as output.

When we change “global” to “nonlocal”:

```
def f():
    nonlocal x
    print(x)
x = 3
f()
```

the program gives an error

```
File "local.py", line 2
    nonlocal x
SyntaxError: no binding for nonlocal 'x' found
```

This means that nonlocal bindings can only be used inside of nested functions. A nonlocal variable has to be defined in the enclosing function scope. If the variable is not defined in the enclosing function scope, the variable cannot be defined in the nested scope. This is another difference to the "global" semantics.

```
def f():
    x = 42
    def g():
        nonlocal x
        x = 43
    print("Before calling g: " + str(x))
    print("Calling g now:")
    g()
    print("After calling g: " + str(x))

x = 3
f()

print("x in main: " + str(x))
```

`nonlocal` keyword that allows you to assign to variables in an outer, but non-global scope.

```
def outside():
    msg = "Outside!"
    def inside():
        msg = "Inside!"
        print(msg)
    inside()
    print(msg)

outside()
```

This will print

```
Inside!
Outside!
```

`msg` is declared in the `outside` function and assigned the value `"Outside!"`. Then in the `inside` function, the value `"Inside!"` is assigned to it. When we run `outside`, `msg` has the value `"Inside!"` in the `inside` function, but retains the old value in the `outside` function.

It is because python didn't actually assigned to the existing `msg` variable, but has created a new variable called `msg` in the local scope of the `inside` that shadows the nae of the variable in the outer scope.

Preventing that behaviour is where the `nonlocal` keyword comes in.

```
def outside():
    msg = "Outside!"
    def inside():
        nonlocal msg
        msg = "Inside!"
        print(msg)
    inside()
    print(msg)

outside()
```

Now, by adding `nonlocal msg` to the top of the `inside`, Python know that when it sees an assignment to `msg`, it should assign to the variable from the outer scope instead of declaring a new variable that shadows its name.

The usage of `nonlocal` is very similar to that of `global`, except that the former is used for variables in outer function scopes and the latter is used for variable in the global scope.

Some confusion might arise about when `nonlocal` should be used. Take the following example,

```
def outside():
    d = {"outside": 1}
    def inside():
        d["inside"] = 2
        print(d)
    inside()
    print(d)

outside()
```

It would be reasonable to expect that without using `nonlocal` the insertion of the `"inside": 2` key-value pair in the dictionary would not be reflected in `outside`.

```
{'inside': 2, 'outside': 1}
{'inside': 2, 'outside': 1}
```

But It is not so, because the dictionary insertion is not an *assignment*, but a method call. In fact, inserting a key-value pair into a dictionary is equivalen to calling the `__setitem__` method on the dictionary object.

```
d = {}
d.__setitem__("inside", 2)
d
```

gives

```
{'inside': 2}
```

Chapter 17

pythonNotes

python notes on important and tricky concepts

Chapter 18

Regular Expressions

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with python's usage of literal backslash for pattern string.

The solution is to use python's raw string notation for regex patterns; backslashes are not handled in any special way in a string literal prefixed with `r`. so `r"\n"` is a two character string containing ´´ and 'n', while `"\n"` is a one character string containing a newline.

Usually patterns will be expressed in Python code using this raw string notation.

Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in the `re` module check if a particular string matches a given regular expression. Regular expressions can be concatenated to form new regular expresssions; if *A* and *B* are both regular expressions, the *AB* is also a regular expression. Repitition qualifiers (`*, +, ?, {m, n}, etc`) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations.

To apply a second repetition to an inner repitition, parentheses may be used. For example `(?:a{6})*` matches any multiple of six `a` characters.

The special characters are:

`.` –> (Dot) In the default mode, this matches any character except a newline. if the `DOTALL` flag has been specified, this matches any character including a new line.

`^` –> (Caret) Matches the start of the string, and in the `MULTILINE` mode also matches immediately after each newline.

`$` –> Matches the end of the string or just before the newline at the end of the string, and in the `MULTILINE` mode also matches before a newline. `foo` matches both `foo` and `foobar`, while the regular expression `foo$` matches only `foo`.

More interestingly, searching for `foo.$` in `foo1\nfoo2\n` matches `foo2` normally, but `foo1` in `MULTILINE` mode; searching for a single `$` in `foo\n` will find two (empty) matches: one just before the newline, and one at the end of the string.

`*` –> Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repititions as are possible. `ab*` will match `a`, `ab` or `a` followed by any number of `b`s.

`+` –> Causes the resulting RE to match 1 or more repititions of the preceding RE. `ab+` will match `a` followed by any non-zero number of `b`s; it will not match just `a`.

`?` –> Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either `a` or `ab`.

`*?, +?, ??`

The `*`, `+`, and `?` qualifiers are all greedy; they match as much text as possible. Sometimes this behaviour is not desired; If the RE `<.*>` is matched against `<a> b <c>`, it will match the entire string, not just `<a>`. Adding `?` after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as few characters as possible will be matched. Using the RE `<.*?>` will only match `<a>`.

`{m}`

Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

`{m, n}`

Causes the resulting RE to match from *m* to *n* repititions of the preceding RE, attempting to match as many repititions as possible. For example, `a{3, 5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omittig *n* specifies an infinite upper bound. As an example, `a{4, }b` will match `aaaab` or a thousand `'a'` characters followed by `b`, but not `aaab`. The comma may not be omitted or the modifier would be confused.

`{m, n}?`

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier.

For example, on the 6-character string `aaaaaa`, `a{3, 5}` will maatch 5 `a` characters, while `a{3, 5}?` will only match 3 characters.

`\`

Either escapes special characters (permitting you to match characters like `*`, `?` and so forth), or signals a special sequence;

Regex Search and Match

Call `re.search(regex, subject)` to apply a regex pattern to a subject string. The function returns `None` if the matching attemp fails, and a `Match` object otherwise.

Since `None` evaluates to `False`, you can easily use `re.search()` in an `if` statement. The `Match` object stores details about the part of the string matched by the regular expression pattern.

re.compile(pattern, flags=0)

compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the following variables, combined using bitwise OR (the | operator)

the sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using `re.compile` and saving the resulting regular expression object for reuse is more efficient when the expression will used several times in a single program.

Matching modes inside the regular expression

Normally, matching modes are specified outside the regular expression. In a programming language, you can pass them as a flag to the regex constructor or append them to the regex literal.

`(?i)` makes the regex case insensitive `re.I`.

match function

This function attempts to match RE **pattern** to string with optional flags.

```
re.match(pattern, string, flags=0)
```

pattern –> This is the regular expression to be matched.

string –> This is the string, which would be searched to match the pattern at the beginning of string.

flags –> you can specify different flags using bitwise OR (|). These are modifiers

The `re.match` function returns a **match** object on success, None on failure. We usegroup(num) or groups() function of **match** object to get matched expression.

method object Method & description

group(num=0) –> This method returns entire match (or specific subgroup num)

groups() –> This method returns all matching subgroups in a tuple (empty if there weren't any.)

Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR(|)

`re.I` `re.IGNORECASE` performs case-insensitive matching.

`re.L` `re.LOCALE` Interprets words according to the current locale. This interpretatioin affects the alphabetic group, as well as word boundary behaviour (or).

`re.M` `re.MULTILINE` Makes `$` match the end of the line (not the just the end of the string) and makes `^` match the start of any line (not just the start of the string.)

`re.S` `re.DOTALL` makes a period (dot) match any character, include newline.

`re.U` Interprets letters according to the Unicode character set. This flag affects the behaviour .

`re.X` `re.VERBOSE` This flag allows us to write regular expressions that look nices and are more readable by allowing you to visually seperate logical sections of the pattern and add comments.

Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?`, `(?:` or `(?P<...>`. When a line contains a `#` that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such `#` through the end of the line are ignored.

```
a = re.compile(r"""\d + # the integral part
                   \.   # the decimal part
                   \d * # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

corresponds to the inline flag `(?x)`.

Chapter 19

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.