Python L3 - Modules, Exceptions, Classes, Threading

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

CS302 2019
Hammond Pearce

Functions

- We have already seen functions in action
- Allow us to define reusable behaviour

```
def sum(a, b):
    output = a + b
    return output

print(sum(4,5))
-> 9
```

Modules

- Modules allow us to group units of behaviour into reusable units
- Modules can be *imported* (keywords `import` and `from`)

```
import time
print(time.time())
time.sleep(20)
print(time.time())
-> 1556157026.4721951
-> 1556157031.4772956
```

Modules

- Modules allow us to group units of behaviour into reusable units
- Modules can be *imported* (keywords `import` and `from`)

```
from time import sleep

sleep(20)
print(time()) #error
```

Modules

- Modules allow us to group units of behaviour into reusable units
- Modules can be *imported* (keywords `import` and `from`)

```
from time import *


sleep(20)
print(time())
```

- ***It’s best to not do this,*** `import *` ***can cause namespace conflicts***

Modules

- Heaps of modules are included with Python by default
  - The “Python Standard Library”
  - This is similar to the “Java Standard Library” and the “C Standard Library” etc
- Other people publish modules too
  - E.g. cherrypy, jinja2
- Code can (and should) be broken up over modules

In-class demo (04.1)

Write a module ‘cs302shapes’ which implements functions for calculating area for

1. *Squares*
2. *Circles*
3. *Triangles*

Then, from a different script, import the module and use the functions

Modules in your project

- It’s a good idea to use modules in your project
- Skeleton code for lab has three modules,
  - Cherrypy
  - Startup / “main”
  - Web handlers
- Good idea to have more, e.g.
  - HTML Templating
  - Break web handlers into UI handlers and API handlers
  - Authentication/security
  - Etc…
- Each module has one “area of responsibility”

Exceptions

- Exceptions are how Python code indicates errors
- Sometimes you know you might get an error, handle it appropriately
    - E.g. file opening fails, type conversion fails

```
try:
    output = “Number of inputs: “ + num
except TypeError:
    output = “Number of inputs: “ + str(num)
```

Exceptions

- Raising an exception will immediately “return” the function

```
def e_fn():
    raise Exception(“error message”)
    return “return message”

try:
    print(e_fn())
except Exception as e:
    print(e)
-> “error message”
```

Exceptions

- Python uses exceptions liberally in its standard library
- You should use them a lot as well
- In a dynamic environment, they ensure that errors are handled properly
    - (As opposed to returning an error message, an exception *must* be caught)

```
try:
    file = open('file.log')
    read_data = file.read()
    file.close()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

In-class demo (04.2)

Write a function which adds two numbers

If either input is negative, throw an Exception

If either input is greater than 10, throw an Exception

Then, write a function which can catch these Exceptions

Finally keyword

- Finally is useful for cleanup operations

```
try:
    raise Exception(“error message”)
except Exception as e:
    print(e)
finally:
    print(“done”)


-> “error message”
-> “done”
```

Finally keyword

- Finally is useful for cleanup operations

```
try:

    #try read file handle

except:

    #file handle error

finally:

    #close file handle
```

With keyword

- `finally` is a bit verbose, and it can be difficult to create buildup/teardown flows
- Introducing: the `with` keyword

```
with open('file.log') as file:
        read_data = file.read()
```

- `with` automatically closes file() for you in this example
- `with` can be used on certain functions that are defined in a compatible manner
    - Check the Python docs

Classes

- Classes allow *objects* which are *units of state and computation* to be defined
    - (Object-oriented programming)
- Based on the C++ model, but simplifed
- Unlike Java, you have the *option* of making something a class
    - You need to decide!
    - Is it structured data that groups together?
    - Does it make sense to think of the data as an object?
    - Do you want/need to create/destroy/alter the data using well-defined methods?

Classes

- Python classes are very similar to methods syntactically!
- Note the “self” argument

```
class Square:

    def setSize(self, h, w):

        self.height = h

        self.width = w

    def getArea(self):

        return self.height * self.width

s = Square()

s.setSize(2,5)

print(s.getArea())

-> 10
```

Classes

- Some special function names

```
class Square:
    def __init__(self, h, w): #constructor
        self.height = h
        self.width = w
    def getArea(self):
        return self.height * self.width


s = Square(2,5)
print(s.getArea())
-> 10
```

Classes

- Attributes defined in the top level are global to all instances

```
class Cat:
    kind = “feline” #common to all instances
    def __init__(self, name):
        self.name = name #only this instance

c1 = Cat(“garfield”)
c2 = Cat(“sylvester”)

c1.kind = “feline”, c1.name = “garfield”
c2.kind = “feline”, c2.name = “sylvester”
```

Classes

- Be careful with mutable global attributes!

```
class Cat:
    favorites = []
    def __init__(self, name, fav):
        self.name = name
        self.favorites.append(fav)

c1 = Cat(“garfield”, “lasagne”)
c2 = Cat(“sylvester”, “tweety bird”)

c1.name = “garfield”, c1.favourites = [“lasagne”, “tweety bird”]
c2.name = “sylvester”, c2.favourites = [“lasagne”, “tweety bird”]
```

Classes

- Be careful with mutable global attributes!

```
class Cat:
    def __init__(self, name, fav):
        self.name = name
        self.favorites = []
        self.favorites.append(fav)

c1 = Cat(“garfield”, “lasagne”)
c2 = Cat(“sylvester”, “tweety bird”)

c1.name = “garfield”, c1.favourites = [“lasagne”]
c2.name = “sylvester”, c2.favourites = [“tweety bird”]
```

Classes

- self is the first argument of a class method
- It provides a reference to the instance
- We define methods of objects similar to functions

```
def methodName(self, arguments...)
```

- We call methods similar to java

```
instance.methodName(arguments...)
```

- As always, be careful with scoping
  - It’s typically the “smallest possible”, but test often!

Classes

- Inheritance is supported

```
class derivedClassName(baseClassName):
```

```
class Cat(Animal):
```

- Python does not have public/private etc,
    - convention is to put _ before private methods and attributes
    - But, it is not enforced

```
class Cat:
    _secret = “a secret known to all cats”
```

- Attributes do not need to be defined at instantiation
    - Python is runtime: Just like variables, attributes can be defined at any time

```
class Cat:
    def __init__(self, name):
        self.name = name


c1 = Cat(“garfield”)
c1.color = “orange” #perfectly fine!
```

- However… as always, do try to keep your code sane!

In-class demo (04.3)

Write a class for a lottery entrant, where each instance has

1. *An ID number,*
2. *a name,*
3. *an age,*
4. *and the number of times they entered*
5. *A method for incrementing the number of entries*

Then,

1. *Make a list of entrants and add some entries loaded from a file (use 03.3 file)*
2. *Deduce the mean and median age of the entrants,*
3. *And choose a random winner (fairly)* (random.choice *might be helpful)*

Threading (a brief introduction)

- Import `threading`
- Logical concurrency
- Make sure you use it usefully
  - Do you need to do things at the same time?
  - Do you need to do something while blocking?
  - How much control do you need over the threads?
  - What order will things be done in?
  - How can you avoid data hazards?
  - Will threading make it slower or faster?
- Threading is “not for beginners”

Threading example

```
import queue, threading, urllib.request #not valid but just saving lines
def get_url(q, url):
    q.put(urllib.request.urlopen(url).read()) #Try to read from the URL

urls=["http://google.com", "http://yahoo.com"]
q = queue.Queue() #Create a Queue object, responses stored here
for u in urls:
    t = threading.Thread(target=get_url, args=(q,u))
    t.daemon = True #lower priority, will shut down when main shuts down

t.start() #Start the thread(s)
print(q.get())
```

Threading

- Remember that threading is logical concurrency
  - For real parallelism / real performance, import `multiprocessor`
- Essentially,
  - Create a Thread object
  - Attach the Thread to a function
    - That function can call other functions / loop / etc
  - Start the thread, use Sleep() if necessary
  - The main thread will continue, use .join() to block
  - Unless other threads are daemons, they will block main thread terminating
- Threads are a little tricky - only use if and when appropriate!

Conclusions

- Modules and classes are a powerful way to structure your code
- Exceptions are a good way to pass error messages around
- Threading can be used in advanced situations