

Lambda and Built-In Functions (cont'd)

- Zip — make an iterator that aggregates elements from each of the iterables
 - Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables
 - The iterator stops when the shortest input iterable is exhausted

```
first_zip = zip([1,2,3], [4,5,6])

list(first_zip) # [(1, 4), (2, 5), (3, 6)]

dict(first_zip) # {1: 4, 2: 5, 3: 6}
```

We could also use the * operator to unpack the list in the following example:

```
five_by_two = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]

list(zip(*five_by_two))

[(0, 1, 2, 3, 4), (1, 2, 3, 4, 5)]
```

Interleaving Strings (kind of tough!)

This challenge is a bit more involved than the others in this section. Do not worry if you can't get it!

Write a function `interleave` that accepts two strings. It should return a new string containing the 2 strings interwoven or zipped together. For example:

```
interleave('hi', 'ha') # 'hhia'
```

```
interleave('aaa', 'zzz') # 'azazaz'
```

```
interleave('lzl', 'iad') # 'lizard'
```

This might seem like an easy task using `zip`, but in fact there are a couple intermediate steps to go from `zip` back to a single string. If you need help, I've written up a basic

exercise.py

```
1 def interleave(str1, str2):
2     return ''.join(''.join(x) for x in zip(str1, str2))
```

Line 1, Column 4 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution

Continue



triple_and_filter

Write a function called **triple_and_filter**. This function should accept a list of numbers, filter out every number that is not divisible by 4, and return a new list where every remaining number is tripled.

exercise.py	<pre>1 ''' 2 triple_and_filter([1,2,3,4]) # [12] 3 triple_and_filter([6,8,10,12]) # [24,36] 4 ''' 5 6 def triple_and_filter(l): 7 return [n*3 for n in l if n % 4 == 0]</pre>	✓ Well done, your solution is correct!
	Line 7, Column 16 All changes saved	Reset code
		Check solution Continue ↩️ 🚀

extract_full_name

Write a function called **extract_full_name**. This function should accept a list of dictionaries and return a new list of strings with the first and last name keys in each dictionary concatenated.

exercise.py	<pre>1 ''' 2 names = [{'first': 'Elie', 'last': 'Schoppik'}, {'first': 'Colt', 'last': 'Steele'}] 3 extract_full_name(names) # ['Elie Schoppik', 'Colt Steele'] 4 ''' 5 6 def extract_full_name(l): 7 return list(map(lambda val: "{} {}".format(val['first'], val['last']), l))</pre>	✓ Well done, your solution is correct!
	Line 6, Column 24 All changes saved	Reset code
		Check solution Continue ↩️ 🚀

Debugging and Error Handling

Objectives:

- Explain common errors and how they occur in Python
- Use pdb to set breakpoints and step through code
- Use try and except blocks to handle errors

Common Types of Errors:

- **SyntaxError** — occurs when Python encounters incorrect syntax (something it doesn't parse)
 - Usually due to typos or not knowing Python well enough

- `NameError` — this occurs when a variable is not defined, i.e. it hasn't been assigned
- `TypeError` — occurs when:
 - An operation or function is applied to the wrong type
 - Python cannot interpret an operation on two data types
- `IndexError` — occurs when the user tries to access an element in a list using an invalid index (i.e. one that is outside the range of the list or string)
- `ValueError` — this occurs when a built-in operation or function receives an argument that has the right type but an inappropriate value
- `KeyError` — this occurs when a dictionary does not have a specific key
- `AttributeError` — this occurs when a variable does not have an attribute

<https://docs.python.org/3/library/exceptions.html>

Raise Your Own Exception:

In Python we can also throw errors using the *raise* keyword. This is helpful when creating your own kinds of exception and error messages

i.e. **raise** `ValueError('invalid value')`

Handling Errors:

In Python, it is strongly encouraged to use *try/except* blocks, to catch exceptions when we can do something about them. Let's see what that looks like.

i.e.

```
try:
    Kristina
except:
    print('You tried to use a variable that was never declared...')
```

What is happening in the example above is that the user is trying to catch every error, which means they are not able to correctly identify "what" went wrong. It is highly discouraged to do this.

Try, Except, Else, and Finally:

When using *try*, *except*, *else*, and *finally*, *try* will run first. If there is an error, the *except* block will run next. If there is no error, the *else* block will run. Lastly, the *finally* block will always run no matter what

Debugging with PDB:

PDB stand for Python Debugger. It is a module that will need to be manually imported in order to have access to it.

To set breakpoints in the code, we can use *pdb* by inserting this line:

```
import pdb; pdb.set_trace()
```

Common PDB command lines:

- L — list
- N — next line
- P — print
- C — continue (finishes debugging)

When utilizing PDB, it's better to use it in one line as shown above for testing purposes only so that it can easily be removed when it's no longer needed, for example:

```
def add_numbers(a, b, c, d):  
    import pdb; pdb.set_trace()  
  
    return a+b+c+d
```

Debugging and Error Handling Exercises

Now that you've learned about debugging and error handling - it's time to practice! See below in the comments for the assignment.

exercise.py

```
1 # Write a function called divide, which accepts two parameters (you can call them num1  
  and num2). The function should return the result of num1 divided by num2. If you do  
  not pass the correct type of arguments to the function, it should return the string  
  "Please provide two integers or floats". If you pass as the second argument a 0,  
  Python will raise a ZeroDivisionError, so if this function is invoked with a 0 as the  
  value of num2, return the string "Please do not divide by zero"  
  
2  
3 # Examples  
4  
5 # divide(4,2) 2  
6 # divide([], "1") "Please provide two integers or floats"  
7 # divide(1,0) "Please do not divide by zero"  
8  
9 def divide(num1, num2):  
10     try:  
11         total = num1 / num2  
12     except TypeError:  
13         return "Please provide two integers or floats"  
14     except ZeroDivisionError:  
15         return "Please do not divide by zero"  
16     return total  
17
```

Line 11, Column 28 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution Continue

Modules

Objectives:

- Define what a module is
- Import code from built-in modules
- Import code from other files
- Import code from external modules using pip
- Describe common module patterns
- Describe the request/response cycle in HTTP
- Use the requests module to make requests to web apps

Why use Modules?

- They keep Python files small
- Reuse code across multiple files by importing
- A module is just a Python file!

Built-in Modules Example:

```
import random

random.choice(['apple', 'banana', 'cherry', 'durian'])
```

If the module isn't referenced in the code, there will be a `NameError` that pops up.

Importing Parts of a Module:

- The **from** keyword lets you import parts of a module
- Handy rule of thumb: only import what you need!
- If you still want to import everything, you can also use the `from Module import *` pattern

Different ways to import:

- `import random`
- `import random as omg_so_random`
- `from random import *`
- `from random import choice, shuffle`
- `from random import choice as gimme_one, shuffle as mix_up_fruits`

Built In Modules Exercise

It's time to get some practice with built-in modules. Here's your mission;

1. Import the `math` module
2. Use `math.sqrt` to find the square root of **15129** and save it to variable called `answer`.

exercise.py

```
1 # Import the math module:
2 import math
3 # Use math.sqrt to find the square root of 15129 and save it to variable called answer:
4 answer = math.sqrt(15129)
```

Line 4, Column 10 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution

Continue

🔍

🗨

Built-In Modules: Slightly Tougher Challenge

Define a function called `contains_keyword` that accepts **any number of string arguments**. It should return `True` if any of the arguments are considered Python keywords (things like `def`, `return`, `if`, etc.) Otherwise it should return `False`. Python has a built-in module called `keyword` that contains a method called `iskeyword`. Import `keyword` and then use `keyword.iskeyword` in your own function to determine if a given string is a keyword.

```
contains_keyword("hello", "goodbye") #False
contains_keyword("def", "haha", "lol", "chicken", "alaska") #True
contains_keyword("four", "for", "if") #True
```

exercise.py

```
1 from keyword import iskeyword as key
2
3 def contains_keyword(*args):
4     for item in args:
5         if key(item): return True
6     return False
```

Line 5, Column 15 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution Continue

Custom Modules:

- You can import from your own code
- The syntax is the same as before
- import from the name of the Python file

Custom Module Exercise

This exercise has two files!!! 🐱 TWO FILES!

Your task is to write a function in the `helpers.py` file, and then call it from the `exercise.py` file. More specifically:

1. In `helpers.py`, define a function called `lucky_number()` that always returns the number 37. That's it. It always returns 37, no matter what.
2. In `exercise.py`, import the `helpers` module. In order for the testing logic to work properly, don't use the `'as'` or `'from'` keywords when importing. Just do a plain old import.
3. From inside `exercise.py`, call the `lucky_number` function you defined in the `helpers` module. Save the result to a variable called `num`.

exercise.py
helpers.py

```
1 #Import your helpers module here. Do not use the 'from' or 'as' syntax, just use a plain
  old 'import'
2 import helpers
3
4
5 #Call the lucky_number function from your helpers module, and save the result to a
  variable called num
6 num = helpers.lucky_number()
```

Line 6, Column 29 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution Continue

External Modules:

- Built-in modules come with Python
- External modules are downloaded from the internet
- You can download external modules using **pip**

pip:

- Package management system for Python
- As of 3.4, comes with Python by default
- `python3 -m pip install NAME_OF_PACKAGE`

Autopep8:

This package is used to clean-up code. In order to use the package, just run the following command:

```
autopep8 --in-place NAME_OF_FILE.py
```

We can use `-a` to tell Autopep8 to be aggressive with making adjustments to the code:

```
autopep8 --in-place -a NAME_OF_FILE.py
```

or

```
autopep8 --in-place -a -a NAME_OF_FILE.py
```

Or add as many `-a` as needed depending on how aggressive the user would like for the package to be when cleaning up their code.

▪ Don't **compare boolean** values to True or False using `==`.

Yes: `if greeting:`

No: `if greeting == True:`

Worse: `if greeting is True:`

The example above is one of the many situations where Autopep8 will make adjustments to make the code cleaner.

The `__name__` variable:

- When run, every Python file has a `__name__` variable
- If the file is the main file being run, its value is `"__main__"`
- Otherwise, its value is the file name

import Revisited:

When you use **import**, Python

- Tries to find the module (if it fails, it throws an error)
- Runs the code inside of the module being imported

Ignoring Code on Import:

- In order to run the code if the file is the main file, we use:

```
if __name__ == "__main__":
```

Making HTTP Requests with Python

HTTP Introduction Objectives:

- Describe what happens when you type a URL in the URL bar
- Describe the request/response cycle
- Explain what a request or response header is, and give examples
- Explain the different categories of response codes
- Compare GET and POST requests

The internet — is basically a bunch of computers that are all in a network connected by tubes, whether it be a fiber optic cable, other wires, or cables connecting computers. These are all ways the data can be transferred.

If we were to open up the Chrome browser and type 'google.com', what exactly happens?

1. DNS Lookup

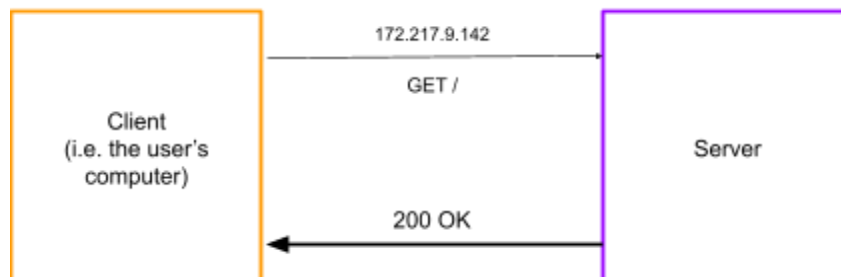
a. What's a DNS Lookup?

A DNS Lookup is essentially a phonebook for the internet



The DNS Server shown above takes the domain names, such as Google.com and turns those domain names into an IP address. The IP address is what we send a request to. Google.com is a human friendly layer.

Requests and Responses



2. Computer makes a REQUEST to a server
3. Server processes the REQUEST
4. Server issues a RESPONSE

This is all called the Request/Response cycle.

The requests and responses contain HTTP Headers, which are essentially the metadata about the requests. It is:

- Sent with both requests and responses
- Provide additional information about the request or response

Header Examples:

Request Headers:

- **Accept** — Acceptable content-types for response (i.e. html, json, xml)
- **Cache-Control** — specify caching behavior
- **User-Agent** — information about the software used to make the request

Response Headers:

- **Access-Control-Allow-Origin** — specify domains that can make requests
- **Allowed** — HTTP verbs that are allowed in requests

Response Status Codes:

- 2xx — SUCCESS
- 3xx — Redirect
- 4xx — Client Error (your fault)
 - 404 is the most common type of 4xx Error, indicating not found
- 5xx — Server Error (not your fault)
 - This indicates that something went wrong on the server side

HTTP Verbs and APIs:

The most fundamental requests within HTTP are GET and POST. These are the two types of requests that a user can make.

- GET
 - Useful for retrieving data or getting information
 - Data passed in query string
 - Should have no "side-effects"
 - Can be cached
 - Can be bookmarked
- POST
 - Useful for writing data (i.e. submitting a new comment on Reddit or posting a photo on Facebook, both of which are things that are being sent to the server)

[illegible]

- Allows you to get data from another application without needing to understand how the application works
- Can often send data back in different formats
- Examples of companies with APIs: GitHub, Spotify, Google

requests Module:

- Let's make HTTP requests from our Python code!
- Installed using pip
- Useful for web scraping/crawling, grabbing data from other APIs, etc.

```
import requests

response = requests.get(
    "http://www.example.com",
    headers={
        "header1": "value1",
        "header2": "value2"
    }
)
```

What's a Query String?:

- A way to pass data to the server as a part of a GET request
- <http://www.example.com/?key1=value1&key2=value2>
 - The above is a way to send data to a server to give more information about a particular request. Each key-value pair is separated by the **&**, i.e.

<https://www.google.com/search?q=cats&oq=cats&aqs=chrome..69i57j46i433j0i433i457j0i402l2j46j46i433l2j0i433l2.788j0j7&sourceid=chrome&ie=UTF-8>

Let's break down the url further to take a look at the key-value pairs:

[https://www.google.com/search?](https://www.google.com/search?q=cats&oq=cats&aqs=chrome..69i57j46i433j0i433i457j0i402l2j46j46i433l2j0i433l2.788j0j7&sourceid=chrome&ie=UTF-8)
q=cats&oq=cats&
aqs=chrome..69i57j46i433j0i433i457j0i402l2j46j46i433l2j0i433l2.788j0j7&
sourceid=chrome&ie=UTF-8

The above shown is the additional information that is being sent as part of the request.

Why does this all matter to us?

- A lot of times, we'll make applications that allow a user to provide some information about what they would like to search for

Object Oriented Programming

Objectives:

- Define what Object-Oriented Programming (OOP) is
- Understand encapsulation and abstraction
- Create classes and instances and attach methods and properties to each

What is OOP?

OOP is about using code to represent or recreate things that exist in the world.

It is a method of programming that attempts to model some process or thing in the world as a **class** or **object**.

- **Class** — a blueprint for objects. Classes can contain methods (functions) and attributes (similar to keys in a dict).
- **Instance** — objects that are constructed from a class blueprint that contain their class's method and properties

Abstraction and Encapsulation:

Why OOP?

- It's mainly about how we structure things and organize things
- With object oriented programming, the goal is to *encapsulate* your code into **logical, hierarchical groupings using classes** so that you can reason about your code at a higher level

Example —

We could have the following entities:

- Game
- Player
- Card
- Deck
- Hand
- Chip
- Bet

Card Deck Possible Implementation (Pseudocode)

Deck {class}

- | | |
|---------------------------|--------------------------|
| • <code>_cards</code> | {private list attribute} |
| • <code>_max_cards</code> | {private int attribute} |
| • <code>shuffle</code> | {public method} |
| • <code>deal_card</code> | {public method} |
| • <code>deal_hand</code> | {public method} |
| • <code>count</code> | {public method} |

***NTS: Python does not actually support true private or public variables or attributes or methods*

Encapsulation — the grouping of public and private attributes and methods into a programmatic class, making **abstraction** possible. i.e.

- Designing the Deck class, I make **cards** a private attribute (a list)
- I decide that the length of the cards should be accessed via a public method called **count()** — i.e. **Deck.count()**

Abstraction — exposing only “relevant” data in a class interface, hiding private attributes and methods (aka the “inner workings”) from users

i.e. of Class:

```
class User:          # this does not necessarily have to be camelcase, but it is
    pass             preferred

user1 = User()
print(type(user1))
```

For the class *User()*, it is a blueprint for what every user should look like. In the example above, every user looks identical and does nothing.

In the line *user1 = User()*, we are instantiating a new user instance, which is the User object. Even if we were to add:

```
user1 = User()
user2 = User()
user3 = User()
user4 = User()
```

All of the users above look the same, but they are not the same thing. They are all their own individual users. Similarly to lists, we could have 4 empty lists. They all look the same, but based on the memory location, they are not.

World's Simplest Class Exercise

Define a class called **Vehicle**. It should be completely empty (just add a `pass` statement inside). After the class is defined, create two instances of Vehicle. Save one to a variable called `car` and another to a variable called `boat`.

exercise.py

```
1 - # define the Vehicle class below:
2 - class Vehicle:
3     pass
4 - # instantiate a new Vehicle and save it to a variable called car:
5 car = Vehicle()
6 - # instantiate a new Vehicle and save it to a variable called boat:
7 boat = Vehicle()
```



Line 7, Column 17 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution

Continue

The `__init__` method:

When it comes to creating a class and making a new instance of a class, Python will automatically look for the `'__init__'` method. You don't ever call it explicitly and it will call itself whenever we make a new vehicle. i.e.

```
class User:
    def __init__(self):
        print("A NEW USER HAS BEEN MADE")

user1 = User()
user2 = User()
user3 = User()

print(user1)      # prints A NEW USER HAS BEEN MADE
print(user2)      # prints A NEW USER HAS BEEN MADE
print(user3)      # prints A NEW USER HAS BEEN MADE
```

This example above is to show that the code runs when the new users are created. For the sake of the example, the print statement within the `__init__` method was to show that the code ran, even when it wasn't called. In Python coding, we don't actually use print statements within the `__init__` method at all.

Within the `__init__` method, we are actually initializing the data that each user has. This is done through the keyword *self*. The ***self*** keyword refers to the specific instance of the user class or whatever class we are working with. (Technically, the parameter doesn't have to be called *self*, but it is the standard and pretty much the only thing anyone will see)

```

class User:
    def __init__(self, first):
        self.name = first

user1 = User("JP")
user2 = User("Kristina")
user3 = User("Athena")

print(user1.name)      # prints JP
print(user2.name)      # prints Kristina
print(user3.name)      # prints Athena

```

Classes in Python can have a special `__init__` method, which gets called every time an instance of the class is created (instantiate).

```

class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

```

Instantiating a Class — creating an object that is an instance of a class called **instantiating** a class

```
v = Vehicle("Honda", "Civic", 2017)
```

In this case, **v** becomes a Honda Civic, a new instance of Vehicle.

`self` — this keyword refers to the current class instance.

- Remember, the parameter does NOT have to be called *self*, but it is the standard and pretty much the only thing anyone sees
- It **must always** be the first parameter to `__init__` and any methods and properties on class instances.

Your First Class - Social Media Comments

It's time to define your own class! Suppose we're creating a social network application where users can comment on posts and photos.

Create a class called `Comment`. Each comment should have the following attributes:

- `username` - the username of the person who created the comment (like "bluethecat")
- `text` - the actual comment itself (like "omg so cute!" or "hahah")
- `likes` - the number of likes the comment has. **Likes should default to 0.**

The following code should work:

```
1 c = Comment("davey123", "lol you're so silly", 3)
```

exercise.py

```
1 # Define the Comment class below:
2
3 class Comment:
4     def __init__(self, username, text, likes=0):
5         self.username = username
6         self.text = text
7         self.likes = likes
```

Line 4, Column 47 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution Continue

What do underscores actually do? What is the significance of the underscores?

There are 3 different ways to use the underscores in Python:

- `_name`
 - When using a single underscore, it is simply a convention. It is a way of telling developers that this variable is a private variable or private property or method.
 - There is no such thing as a private attribute or private method in Python. There are other languages that support private methods, private properties or attributes where it was not accessible outside of the class. However, this is a way for developers to indicate that it is only intended for internal use within the class
- `__name`
 - This is what is called name mangling
 - The sole purpose of it is to make the method or attribute particular to the class
- `__name__`
 - It is a convention that should be respected
 - It's possible to define your own, but you shouldn't
 - They're used for Python specific methods like `init`, that we will then define on our own

Adding Instance Methods:

******Whenever utilizing instance methods that are dunder methods, they are usually placed at the top.

Instance attributes are just like pieces of data that are associated with each individual user.

Instance methods —

```
class User:
    def __init__(self, first, last, age):
        self.first = first
        self.last = last
        self.age = age

    def full_name(self):
        return f"{self.first} {self.last}"

    def initials(self):
        return f"{self.first[0]}.{self.last[0]}."

user1 = User("Blanca", "Lopez", 42)
print(user1.full_name())          # returns Blanca Lopez
print(user1.initials())          # returns B.L.
```

Bank Account OOP Exercise

Define a new class called `BankAccount`.

- Each `BankAccount` should have an `owner`, specified when a new `BankAccount` is created like `BankAccount("Charlie")`
- Each `BankAccount` should have a `balance` attribute that **always** starts out as `0.0`
- Each instance should have a `deposit` method that accepts a number and adds it to the

exercise.py

```
1 # Define Bank Account Below:
2 class BankAccount:
3     def __init__(self, name):
4         self.owner = name
5         self.balance = 0.0
6
7     def getBalance(self):
8         return self.balance
9
10    def withdraw(self, amount):
11        self.balance -= amount
12        return self.balance
13
14    def deposit(self, amount):
15        self.balance += amount
16        return self.balance
```

Line 1, Column 1 All changes saved

Reset code

✓ Well done, your solution is correct!

Check solution

Continue

