Head First python

1. There's also no notion of the familiar edit-compile-link-run process. With Python, you edit your code and save it, and run it *immediately*.
2. **interpreter** is the technology that runs your Python code. Rather confusingly, this interpreter is also known by the name "Python."
3. **imports** - uses some preexisting functionality from Python's **standard library,** which is a large stock of software modules providing lots of prebuilt (and high-quality) reusable code.
4. **os module**: which provides a platform-independent way to interact with your underlying operating system - function, getcwd, which—when invoked—returns your *current working directory*.
5. From os import getcwd getcwd()
6. The **standard library** supplys reusable modules that help you with manipulating ZIP archives, to sending emails, to working with HTML.
    1. includes a web server, as well as the popular *SQLite* database technology.
    2. Python prides itself on being cross-platform, in that code written on one platform can be executed (generally unaltered) on another
    3. The sys module exists to help you learn more about your interpreter's identity of your underlying operating system, by first importing the sys module, then accessing the platform attribute:
    4. Import sys sys.platform sys.version
    5. Import os os.environ os.getenv('HOME')
    6. Import datatime datetime.date.today() datetime.date.today().month
    7. datetime.date.isoformat(datetime.date.today())
    8. Import time time.strftime("The hour is %H and minute is %M")
7. https://pypi.org/. Python package index
8. Python also has some powerful built-in **data structures**
9. lists in Python can contain *any* data of *any* type
10. In Python, variables pop into existence the first time you use them, and **their type does not need to be predeclared**.
11. IN operator checks if one thing is *inside* another list. returns either True or False.
12. Python uses **indentation** to demarcate a block of code, which Python programmers prefer to call **suite and** doesn't use curly braces
13. colon character (:), which is used to introduce a suite that's associated with any of Python's control statements (such as if, else, for, and the like).
14. If a number of conditions need to be checked as part of an if statement, Python provides **elif** as well as **else**. You can have as many elif parts (each with its own suite) as needed.
    1. If today == 'Saturday':
        1. Print ("Party")

    2. Elif today == 'Sunday':
        1. Print("Recover")
    3. Else:
        1. print("work");
15. for i in [1,2,3]:
16. ... print(i)
17. for char in "Hello":
18. ... print(char)
19. Iterate specific number of times
20. For I in range(40)
21. import statement can be used *in two ways* - imports a named function into our program's **namespace**, (The notion of a namespace is important in Python, as it defines the context within which your code runs.
22. The second way to use import is to just import the module, as we did when experimenting with the time module. When we import this way, we have to use the dot-notation syntax to access the module's functionality, as we did with time.sleep().
23. **Generating Random Integers with Python**

Shell functions

1. **dir(random)**
2. **M**

LIST

1. *widely applicable* data structures: **lists**, **dictionaries**, **tuples**, and **sets**
2. **A variable takes on the type of the value assigned.**
3. **Everything is an object in Python, and any object can be assigned to a variable.**
4. All data values in Python are objects, can have **state** (attributes or values) and **behavior** (methods).
5. You can certainly program Python in an object-oriented way using classes, objects, instancesbut you don't have to, you do not need to start with a class when first creating code in Python: you just write the code you need.
6. car_details = ['ford','Mustang',2.2,100]
7. List1 = ['list1',['a','b','c']]

Four Built-in Data Structures

1. **List: an ordered mutable collection of objects**
    - list in Python is very similar to the notion of an **array** ,an indexed collection of related objects,
    - lists are **dynamic** in Python, in that they can grow (and shrink) on demand.
    - Lists are also heterogeneous, in that you do not need to predeclare the type of the object you're storing—you can mix'n'match objects of different types in the one list if you like.
    - Lists are **mutable**, in that you can change a list at any time by adding,

removing, or changing objects.
  - list is an example of a **mutable** data structure,
2. **Tuple: an ordered immutable collection of objects**
   - A tuple is an immutable list. once you assign objects to a tuple, the tuple cannot be changed under any circumstance.
     It is often useful to think of a tuple as a constant list.

3. **An Unordered Data Structure: Dictionary**
   - If keeping your data in a specific order isn't important to you, but structure is, a choice of two unordered data structures: **dictionary** and **set**.
   - **Dictionary: an unordered set of key/value pairs**
   - dictionary allows you to store a collection of key/value pairs, each unique **key** has a **value** associated with it in the dictionary,
   - Dictionaries are unordered and mutable.

4. **Unordered set of unique object that Avoids Duplicates: Set**
   - remove duplicates quickly from any other collection. And don't worry if the mention of sets has you recalling high school math class and breaking out in a cold sweat. Python's implementation of sets can be used in lots of places.
   - sets let you perform unions, intersections, and differences
   - can grow (and shrink) as needed.

Program
- Create a list of vowels aeiou and then check if any word contains these vowels
- List methods : Remove , pop, insert , extend
- remove method removes the first occurrence of a specified data value from a list.
- **pop: takes an optional index value as its argument**
  - removes *and returns* an object from an existing list based on the object's index value.
  - If you invoke pop without specifying an index value, the last object in the list is removed and returned.
  - If a list is empty or you invoke pop with a nonexistent index value, the interpreter *raises an error* (more on this later).
- **extend: takes a list of objects as its sole argument**
  - Takes a second list and adds each of its objects to an existing list.
  - This method is very useful for combining two lists into one:
- **insert: takes an index value and an object as its arguments**
  - insert method inserts an object into an existing list *before* a specified index value.
  - This lets you insert the object at the start of an existing list or anywhere within the list. It is not possible to insert at the end of the list, as that's what the append method does:

How to Copy a Data Structure

1. assignment operator is not the way to copy one list to another, using assignment a **reference** to the list is *shared* among first and second.
2. First = [1,2,3,4]
3. Second = first
4. Both first and second point to same object
5. second=first.copy()
6.

———————————————————

DICTIONARY

1. Set of key value pairs
2. dictionaries understand the square bracket dict1['name']
3. **DICTIONARY KEYS MUST BE INITIALIZED**
4.

Tuples Are Immutable

————Functions and Modules

1. Take some lines of code, give them a name, and you've got a function (which can be reused).
2. Take a collection of functions and package them as a file, and you've got a **module** (which can also be reused)
3. Python supports **modularity**, in that you can break large chunks of code into smaller, more manageable pieces
4. **Functions introduce two new keywords: def and return**
    1. The def keyword names the function (shown in blue), and details any arguments the function may have.
    2. return keyword is optional, and is used to pass back a value to the code that invoked the function.
5. **Functions can accept argument data**
6. **Functions contain code and (usually) documentation**
    1. Code is indented one level beneath the def line
    2. ways to add comments to code: using a triple-quoted string
    3. using a single-line comment, which is prefixed by the # symbol (and shown in red, below).
7. The Python interpreter does not force you to specify the type of your function's arguments or the return value.
8. Python lets you send any *object* as a argument, and pass back any *object* as a return value.

————bool————

```
>>> bool()
False
>>> bool(0)
False
>>> bool(0.1)
True
>>> bool('')
False
>>> bool([])
False
>>> bool({})
False


->>> list()
[]

>>> set()
set([])

>>> dict()
{}

>>> tuple()
()

>>> list([1,2,3,'a'])
[1, 2, 3, 'a']


>>> set({1,2,3,'a'})
set(['a', 1, 2, 3])


>>> list([1,2,3,'a','a'])
[1, 2, 3, 'a', 'a']


>>> set({1,2,3,'a','a'})
set(['a', 1, 2, 3])

>>> dict({'key':'value','key':'value'})
{'key': 'value'}

>>> set((1,2,3,4,4))
set([1, 2, 3, 4])
```

```
>>> tuple((1,2,3,4,4))
(1, 2, 3, 4, 4)
>>> set({1,2,3,4,4})
set([1, 2, 3, 4])
>>>
```

————————Annotation

1. **annotations** (also known as *type hints*) document—in a standard way—the return type, as well as the types of any arguments. Keep these points in mind:
2. **Function annotations are informational**
   They provide details about your function, but they do not imply any other behavior (such as type checking).
   1. The first annotation states that the function expects a string as the type of the word argument (:str), while the second annotation states that the function returns a set to its caller (-> set):
   2. the interpreter **won't** check that the function is always called with a string, nor will it check that the function always returns a set.
   3. **Use annotations to help document your functions, and use the "help" BIF to view them.**
   4.

————**Specifying Default Values for Arguments**

1. Any argument to a Python function can be assigned a default value, which can then be automatically used if the code calling the function fails to supply an alternate value.
2. The mechanism for assigning a default value to an argument is straightforward: include the default value as an assignment in the function's def line.

Here's search4letters's current def line:
def search4letters(phrase:str, letters:str) -> set:

————————Keyword arguments

```
print(search4letters(letters='s',phrase='a string' ))
```

Share your functions in modules.

1. create a **module** that contains a single, canonical copy of any functions you want to share.
2. a module is any file that contains functions.
3. Once your module exists, making its contents available to your programs : import the module using Python's import statement.

4. three main locations the interpreter searches when looking for a module. These are:
    1. **Your current working directory**
    2. **Your interpreter's site-packages locations**
        1. These are the directories that contain any third-party Python modules you may have installed (including any written by you).
    3. **The standard library locations**
    These are the directories that contains all the modules that make up the standard library.

USING "SETUPTOOLS" TO INSTALL INTO SITE-PACKAGES

1. **Create a distribution description**: This identifies the module we want setuptools to install.
    1. *Create a distribution description*, we need to create two files that we'll place in the same folder as our vsearch.py file.
    2. The first file, which must be called setup.py, describes our module in some detail.
    3. It contains two lines of Python code: the first line imports the setup function from the setuptools module, while the second invokes the setup function.
    4. The setup function accepts a large number of arguments, many of which are optional. The most important arguments are highlighted; the first names the distribution, while the second lists the .py files to include when creating the distribution package:

```
from setuptools import setup

setup(
    name = 'vsearch',
    version = '1.0',
    description = 'Head First Python Search tools',
    author = 'yash',
    author_email = 'yashniga@gmail.com',
    url = 'gmail.com',
    py_modules=['vsearch']
)
```

    5. In addition to setup.py, the setuptools mechanism requires the existence of one other file—a "readme" file—into which you can put a textual description of your package.
2. **Generate a distribution file** :create a shareable distribution file to contain our module's code.
    1. This is simply a file file containing our code
3. **Install the distribution file** : install the distribution file (which includes our module) into site-packages.
    1. Python3 setup.py sidst

```
MacBook-Pro:mymodules yashnigam$ python3 setup.py sdist
running sdist
running egg_info
creating vsearch.egg-info
writing vsearch.egg-info/PKG-INFO
writing dependency_links to vsearch.egg-info/dependency_links.txt
writing top-level names to vsearch.egg-info/top_level.txt
writing manifest file 'vsearch.egg-info/SOURCES.txt'
reading manifest file 'vsearch.egg-info/SOURCES.txt'
writing manifest file 'vsearch.egg-info/SOURCES.txt'
warning: sdist: standard file not found: should have one of
README, README.rst, README.txt, README.md

running check
creating vsearch-1.0
creating vsearch-1.0/vsearch.egg-info
copying files to vsearch-1.0...
copying setup.py -> vsearch-1.0
copying vsearch.py -> vsearch-1.0
copying vsearch.egg-info/PKG-INFO -> vsearch-1.0/vsearch.egg-
info
copying vsearch.egg-info/SOURCES.txt -> vsearch-1.0/
vsearch.egg-info
copying vsearch.egg-info/dependency_links.txt -> vsearch-1.0/
vsearch.egg-info
copying vsearch.egg-info/top_level.txt -> vsearch-1.0/vsearch.egg-
info
Writing vsearch-1.0/setup.cfg
creating dist
Creating tar archive
removing 'vsearch-1.0' (and everything under it)
```

2. three files have been combined into a single **distribution file**, This is an installable file that contains the source code for your module and, in this case, is called vsearch-1.0.zip.
3. You'll find your newly created ZIP file in a folder called dist, which has also been created by setuptools under the folder you are working in (which is mymodules in our case).

4. Install using pip
    1. Python 3 -m pip install vsearch-1.0.tar.gz

```
python3 -m pip install dist/vsearch-1.0.tar.gz
Processing ./dist/vsearch-1.0.tar.gz
Installing collected packages: vsearch
  Found existing installation: vsearch 1.0
    Can't uninstall 'vsearch'. No files were found to uninstall.
  Running setup.py install for vsearch ... done
Successfully installed vsearch-1.0
```

You are using pip version 10.0.1, however version 20.2b1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.

GIVING YOUR CODE AWAY (A.K.A. SHARING)

1. **Python programmer can also use pip to install your module.**
2. To share your module formally, you can upload your distribution file to Python's centrally managed web-based software repository, called PyPI (pronounced "pie- pee-eye," and short for the *Python Package Index*).
3. This site exists to allow all manner of Python programmers to share all manner of third-party Python modules.

Pass by value and pass by refrence

1. Depending on the situation, Python's function argument semantics support **both** call-by-value *and* call-by-reference.
2. variables in Python are **object references**.
3. value stored in the variable is memory address of the value, not its actual value, It's this memory address that's passed into a function, not the actual value.
4. This means that Python's functions support what's more correctly called *by-object-reference call semantics*.
5. interpreter looks at the type of the value referred to by the object reference (the memory address) and, if the variable refers to a **mutable** value, call-by-reference semantics apply.
6. If the type of the data referred to is **immutable**, call-by- value semantics kick in. Consider now what this means for our data.
7. Lists, dictionaries, and sets (being mutable) are always passed into a function by reference— any changes made to the variable's data structure within the function's suite are reflected in the calling code.
8. Strings, integers, and tuples (being immutable) are always passed into a function by value— any changes to the variable within the function are private to the function and are not reflected in the calling code.
9. arg = arg * 2

How come this line of code appeared to change a passed-in list within the function's suite, but when the list was displayed in the shell after invocation, the list hadn't changed (leading Tom to believe—incorrectly—that all argument passing conformed to call-by-value)? On the face of things, this looks like a bug in the interpreter, as we've just stated that changes to a mutable value are reflected back in the calling code, but they aren't here. That is, Tom's function *didn't* change the numbers list in the calling code, even though lists are mutable. So, what gives?
To understand what has happened here, consider that the above line of code is

an **assignment statement**. Here's what happens during assignment: the code to the right of the = symbol is executed *first*, and then whatever value is created has its object reference assigned to the variable on the left of the = symbol. Executing the code arg * 2 creates a *new* value, which is assigned a *new* object reference, which is then assigned to the arg variable, overwriting the previous object reference stored in arg in the function's suite. However, the "old" object reference still exists in the calling code and its value hasn't changed, so the shell still sees the original list, not the new doubled list created in Tom's code. Contrast this behavior to Sarah's code, which calls the append method on an existing list. As there's no assignment here, there's no overwriting of object references, so Sarah's code changes the list in the shell, too, as both the list referred to in the functions' suite and the list referred to in the calling code have the *same* object reference.
With our mystery solved, we're nearly ready for Chapter 5. There's just one outstanding issue.

PyTest

1. **pytest**, which is a *testing framework* that is primarily designed to make the testing of Python programs easier.
2. No matter what type of tests you're writing, **pytest** can help. And you can add plug-ins to **pytest** to extend its capabilities.
3. One such plug-in is **pep8**, which uses the **pytest** testing framework to check your code for violations of the PEP 8 guidelines.
4. **How PEP 8–Compliant Is Our Code?**

Let's build a webapp.

1. take our search4letters function and make it accessible over the Web, enabling anyone with a web browser to access the service provided by our function.
2. server-side web application is a **web application framework**, which provides a set of general foundational technologies upon which you can build your webapp.
3. **Let's Install Flask**
4. the Python community maintains a centrally managed website for third-party modules called **PyPI** (short for *the Python Package Index*), which hosts the latest version of Flask (as well as many other projects).

How Does Flask Work?

1. Flask provides a collection of modules that help you build server-side web applications. It's technically a *micro* web framework, in that it provides the minimum set of technologies needed for this task. This means Flask is not

as feature-full as some of its competitors—such as **Django**, the mother of all Python web frameworks—but it is small, lightweight, and easy to use.
2. **Django** is a hugely popular web application framework within the Python community. It has an especially strong, prebuilt administration facility that can make working with large webapps very manageable. It's overkill for what we're doing here, so we've opted for the much simpler, but more lightweight, **Flask**.

Code
```
from flask import Flask

app= Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello World from flask'

app.run();
```

1. The __name__ value is maintained by the Python interpreter and, when used anywhere within your program's code, is set to the name of the currently active module. It turns out that the Flask class needs to know the current value of __name__ when creating a new Flask object, so it must be passed as an argument, which is why we've used it here (even though its usage does look *strange*).
2. Note that __name__ is two underscore characters followed by the word "name" followed by another two underscore characters, which are referred to as "double underscores" when used to prefix and suffix a name in Python code. You'll see this naming convention a lot in your Python travels, and rather than use the long-winded: "double underscore, name, double underscore" phrase, savvy Python programmers say: "dunder name," which is **shorthand for the same thing**. As there's a lot of double underscore usages in Python, they are collectively known as "the dunders," and you'll see lots of examples of other dunders and their usages throughout the rest of this book.

Decorating a Function with a URL

1. **A function decorator adjusts the behavior of an existing function (without changing the function's code).**
2. Python's decorator syntax take inspiration from Java's annotation syntax, as well as the world of functional programming.
3. decorators allow you to take some existing code and augment it with additional behavior as needed.
4. Although decorators can also be applied to classes as well as functions, they are mainly applied to functions, which results in most Python programmers referring to them as **function decorators**.

5. @app.route('/')
6. Flask's route decorator is available to your webapp's code via the app variable, which was created on the previous line of code.
7. The route decorator lets you associate a URL web path with the Python function that immediately follows the route decorator in your code. In this case, the URL "/" is associated with the function defined on the very next line of code, which is called hello. The route decorator arranges for the Flask web server to call the function when a request for the "/" URL arrives at the server. The route decorator then waits for any data returned from the decorated function before returning the output to the server, which then returns it to the waiting web browser.
8. It's not important to know how Flask (and the route decorator) does all of the above "magic." What is important is that Flask does all of this for you, and all you have to do is write a function that produces the output you require. Flask and the route decorator then take care of the details.
9. With the route decorator line written, the function decorated by it starts on the next line. In our webapp, this is the hello function, which does only one thing: returns the message "Hello world from Flask!" when invoked:

```
@app.route('/')
def hello() -> str:
    return 'Hello World from flask
```

10. With the route decorator line written, the function decorated by it starts on the next line. In our webapp, this is the hello function, which does only one thing: returns the message "Hello world from Flask!" when invoked:
11. The final line of code takes the Flask object assigned to the app variable and asks Flask to start running its web server. It does this by invoking run:

TEMPLATES UP CLOSE

1. Template engines let programmers apply the object-oriented notions of inheritance and reuse to the production of textual data, such as web pages.
2. A website's look and feel can be defined in a top-level HTML template, known as the **base template**, which is then inherited from by other HTML pages.
3. If you make a change to the base template, the change is then reflected in *all* the HTML pages that inherit from it.
4. The template engine shipped with Flask is called *Jinja2*, and it is both easy to use and powerful.
5. Here's the base template we'll use for our webapp. In this file, called base.html, we put the HTML markup that we want all of our web pages to share.
6. We also use some Jinja2-specific markup to indicate content that will be supplied when HTML pages inheriting from this one are rendered (i.e., prepared prior to delivery to a waiting web browser). Note that markup appearing between {{ and }}, as well as markup enclosed

between {% and %}, is meant for the Jinja2 template engine: we've highlighted these cases to make them easy to spot:

base template base.html

```
<!doctype html>
<html>

<head>
   <!-- This is jinja 2 directive its value will be provided before
rendering -->
   <title> {{ the_title }} </title>
   <link rel="stylesheet" href="static/hf.css" />
</head>

<body>
   <!-- This jinja 2 named block should be rendered before
rendering, by actual html code -->
   {% block body %}

   {% end block %}
</body>

</html>
```

7. With the base template ready, we can inherit from it using Jinja2's extends directive.
8. When we do, the HTML files that inherit need only provide the HTML for any named blocks in the base. In our case, we have only one named block: body.
9. Here's the markup for the first of our pages, which we are calling entry.html.
10. This is markup for an HTML form that users can interact with in order to provide the value for phrase and letters expected by our webapp.
11. Note how the "boilerplate" HTML in the base template is not repeated in this file, as the extends directive includes this markup for us. All we need to do is provide the HTML that is specific to this file, and we do this by providing the markup within the Jinja2 block called body:

An html file entry.html utilising base.html

```
{. % extends 'bases.html'  % }
{. % block body % }
{. %. end vlock%}
```

Rendering Templates from Flask

1. Flask comes with a function called render_template, which, when provided with the name of a template and any required arguments, returns a string of HTML when invoked.
2. To use render_template, add its name to the list of imports from the flask module (at the top of your code), then invoke the function as needed.
3.

Handling Posted Data

1. As well as accepting the URL as its first argument, the @app.route decorator accepts other, optional arguments.
2. One of these is the methods argument, which lists the HTTP method(s) that the URL supports. By default, Flask supports GET for all URLs. However, if the methods argument is assigned a list of HTTP methods to support, this default behavior is overridden.
3. To have the */search4* URL support POST, add the methods argument to the decorator and assign the list of HTTP methods you want the URL to support. This line of code, below, states that the */search4* URL now only supports the POST method (meaning GET requests are no longer supported):

Using Request Data in Your Webapp

1. To use the request object, import it on the from flask line at the top of your program code, then access the data from request.form as needed.

We know from the information on the last page that we can access the phrase entered into the HTML form within our code as request.form['phrase'], whereas the entered letters is available to us as request.form['letters']. Let's adjust the do_search function to use these values (and remove the hardcoded strings):

Replacing values in results.html

1. There are four jinja2 variables: the_title, the_phrase, the_letters, and the_results.
2. Take another look at the do_search function's code (below), which we are going to adjust in just a moment to render the HTML template shown above. As you can see, this function already contains two of the four variables we need to render the template (and to keep things as simple as possible, we've used variable names in our Python code that are similar to those used in the Jinja2 template):

Preparing Your Webapp for the Cloud

1. One popular service is cloud-based, hosted on AWS, and is called *PythonAnywhere*. We love it over at *Head First Labs*.
2. *PythonAnywhere* likes to control how your webapp starts, this means *PythonAnywhere* assumes responsibility for calling app.run() on your behalf, which means you no longer need to call app.run() in your code.
3. It would be nice if there were a way to selectively execute code based on whether you're running your webapp locally on your computer or remotely on *PythonAnywhere*...
4. This three-line program begins by displaying a message on screen that prints the currently active namespace, stored in the __name__ variable. An if statement then checks to see whether the value of __name__ is set to __main__, and—if it is—another message is displayed confirming the value of __name__ (i.e., the code associated with the if suite executes):

1. if your program code is executed *directly* by Python, then __name__ will be __main__
2. If your program code is imported as a module value of __name__ is not __main__, but the name of the imported module (dunder in this case).
3. To make our code work in cloud as well as local

```
if __name__ == '__main__':
    app.run(debug=True)
```

1.