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?**