Head First – C1 – The basics

List is a type of array but ~~bar~~ is not declared unlike arrays.

standard library in python contains lots of of modules (random, datetime, time) which has a lot of functions.

Suites are the indented blocks of code. which may inclued more suites.

datetime.datetime.today().minute gives the minute of this instant.

Sleep(x) of the time module makes the program pause for x seconds.

randint(a,b) of random module selects a random integer in the range of a and b inclusive.

help(random.randint) displays knowledge neccessary from the pyfho docs.

dir function list an object's attributes.

range(start, stop, step)

Chapter 2 - List - Data

Everything in python is an object.

python uses duck-typing unlike other languages, meaning, we do not need to specify the type of an object when assigning it to a variable.

There are four built in Data structures -
- List, tuple, dictionary and set
  ordered unordered

Properties of list - ordered &
- mutable (more objects can be added/removed)
- dynamic (can grow/shrink in size)
- [...], - also Heterogenous.

Arrays are not dynamic.

Properties of tuple - ordered
- immutable (no objects can be added or removed)
- (...)

Properties of dictionary - unordered
- ~~is~~ contains values assigned to keys
- dynamic
- mutable.

Hepaximin

Properties of sets - Avoids duplicates
- dynamic
- unordered

The len() built in function reports on the size of a list.

.append() method adds a value to the list.

.remove() method checks if a value is in a list and removes it if its there. Else pulls out an error.

Note- if is not used indexically.

.pop() method removes and returns an object from an existing list based on the object's index value.

If no index value is specified, it will remove the last value.

.extend() takes a list of objects and adds it to another list.

.insert(a, b) takes a value b and places it in index a of the list.

We can use the .copy() method to copy a list.

Example ⇒ second = [1, 2, 3, 4]

third = second.copy()

~~>>> third >>> [1, 2, 3, 4]~~

- third.append(5)
- print(third) ↵ [1, 2, 3, 4, 5]

Properties of sets - Avoids duplicates

Unlike othe programming languages, python lets us access the lish relative to each endj postive index goes from left to right, and negative goes from right to left.

List like range, use the [start:stop:step] method.
Note - they take indexes as parameters.

.join (plist[1:3]) method takes [1:3] of the indexes of plist and joins it to another list/variable.

We do not have to show the for loop where the list starts and ends.

They also understand [start:stop:step].

Hepaximin

Chapter - 3 - Structured data

Dictionaries are used to store a collection of key value pairs.

insertion
As the order is not maintained, index searching does not work.
Instead, use key searching.

Example - Person 3 = {'Name': 'Ford Prefect',
'Gender': 'Male',
'Occupation': 'Researcher'
'Home planet': 'Betelgeuse seven'}

Person 3 = dictionary, key → 'Name', value → 'Ford Prefect'

```
>>> Person3['Name']
'Ford Prefect'
```

We can also add a key-value pair at Runtime.
Example - person3['Age'] = 33

Iterating over a for loop only returns the keys, that too in an unordered manner.
Hence, we use sorted() function to sort and .items() to get both key value pairs.

eg- for k,v in sorted(found.items()):

sets do not allow duplicates. The main three other reasons they are used is because of their functions
- Union
- Intersection
- Difference

set1 = {'....', }

Tuples are non-dynamic. Tuple1 = ('...', '...')

Tuples are Immutable.

Only use a tuple if the data never changes.

use a comma to create even a single object tuple.

('a',)

As everything is an object in python, we can store whatever data structure inside other data structures.

In complex data-structures use pretty-print to print in a readable manner. using pprint().

- Import pprint first.

Chapter-4 - Reuse data

Functions are used to re-use chunk of code.

We use def followed by the name of the function to create a function.

Function can return, True, False or values.

Functions can take arguments to evaluate/process the code.

Functions can return multiple values in functions.
to get our way-around, we put the data in a data-structure and return that.

bool(x) returns a ~~yes~~ true or false ~~or~~ considering the x.

Annotations are used in functions to make it easy for the code-reader by letting them know information about the function. (They do not specify the data type for python).

eg - def Search4vowels(word: str) → set:
(word: str → arg, → set → returns a set.)

using >>> help(search4vowels) will explain it.

python's triple ~~code~~quote strings can be used to add multiple lines of comments known as docstring.

To test a program's function, we can either take an input, and the pass it to the function (1) or just pass it directly to the function.

(1) input = ( ), search4letters(input)

(2) search4letters ~~(input())~~ ("galaxy", "xyz") (search4letters("galaxy", "xyz")).

Default value is used for an until a/@nguument. is provided.
For example,

```
def search4letters(phrase: str, letters: str = 'aeiou') -> set:
```

setting a default value.

⇒ A default value will not be used if an argument for letters is passed on.

Using Default value makes our programs more generic.

To invoke a function we use keyword arguments which can be of two type -

① Positional assignment → search4letters('galaxy', 'sky')
Phrase letters

② keyword assignment → search4letters(letters='galaxy', phrase='sky')

A module is any file that contains functions. We create a module to share functions.

To create a shareable module follow 3 steps -

1) Create a setup.py file in the same folder as ~~search4letters.~~ vsearch.py.

2) Generate a distribution file using cmd and "py-3 setup.py sdist" in the same folder.

3) use py-3 -m pip install vsearch-1.0.zip.

Chapter 5- Building a webapp

Python Community maintains a centrally managed website for third party modules called PyPI. (Python Package Index)

To install a module from PyPI, we open a cmd and type - ~~C:>~~ C:\> Py -3 -m pip install .... (module name)

To use a module we go to our Ide, and import the module. - from flask import Flask.
(module name) (function.)

example basic code -

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

Create an instance of flask object and assign it to app.

URL decorators.

Decorators helps us adjest a function's behaviour without changing the function's code.

Route function creates a URL and any request recieved from the URL recieves 'Hello world from Flask' as response.

Flask ~~testlets~~ lets us display the output of our created function in a webpage.

We can generate our own URL by writing our desired Link in .route('/....'). example - .route('/search4')

GET Method - The method by which someones requests information from the web server. GET Method is always given to Flasks unless other methods are defined. If so, the we have to mention GET Method too.

HTTP

To add^ methods that we want our URL to support -

@app.route('/search4', methods = ['POST', 'GET'])

Debugging mode automatically restarst the webapp everytime our code changes.

to

us chapter,

econds.

ss.

s its

appending)

which

Python Co
Son thind

To install

To switch to debugging use
app.run (debug = True) instead of
app. run (app is the name of the
flask),

and typ # ~~Flask comes in~~ Request is a built in
object that provides easy access to posted data.

To use
the r that provides access to a HTML form's data posted
It contains a dictionary attribute called form
from the browser.

To use request function, just use -

Phrase = request. form ['Phrase']
letters = request. form ['letters']

Chapter - 6 - Manipulating data

From the webapp we created in previous chapter, we can log the data requested to keep records.

Python supports Open, Process, Close process.

⇒ This technique lets us open a file, process its data in some way (reading, writing or appending) and then close the file when we are done which saves our changes.

⇒ To open a file, we open windows command line and write py -3.

*then we write todos = open ('todos.txt', 'a')

todos → variable.

'todos.txt' → name of file.

'a' → a for append mode.

* Now that the file is open, we can use print() function to write on it.

- print ('Feed the Cat', file = todos)

file = todos → specify

* close the file once we are done - todos.close()

To read from a file, use "tasks = open ('todos.txt')" then print out using for loop. for chore in tasks:
    Print (chore)
tasks.close()

'r' stands for reading mode and it's also the default mode of open().

'w' stands for writing mode, and delete its previous contents.

'a' stands for appending mode which keeps the previous data and adds more data to it.

'x' opens a new file for writing.

"with" is a better version of Open, Process, close as it closes the file by itself once the task is done preventing data loss.
How to use "with" - with open ('todos.txt')as tasks:
    for chore in tasks:
        Print (chore, end = '')

If we are ~~create~~ a to save data from the webapp, we can create a function which writes on the ~~todos.txt, and not~~ log.

```
# def log_request(req: 'flask_request', res: str) -> None:
        with open('vsearch.log', 'a') as log:
            print(req, res, file=log)

# Make sure to invoke the function from inside
  the do_search function. log_request(request, result)
```

Chapter-5- final code

```
from flask import Flask, render_template, request
from vsearch import search4letters.

app = Flask (__name__)

@app.route ('/search4', methods = ['POST'])
def do_search () -> 'html':
    phrase = request.form ['phrase']
    letters = request.form ['letters']
    title = 'Here are your results:'
    results = str (search4letters (phrase, letters))
    return render_template ('results.html',
                            the_title = title,
                            the_phrase = phrase,
                            the_letters = letters,
                            the_results = results,)

@app.route ('/')
@app.route ('/entry')
def entry_page () -> 'html':
    return render_template ('entry.html',
                            the_title = 'Welcome to... web!')

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

Chapter-6 - final-code

```
...lines 1-3
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res,
              file=log, sep='|')

...lines 4-19
@app.route('/viewlog')
def view_thelog() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)

...lines 20-21
```

Chapter-7 - Using a database

Using a database to store logs and information is more preferred over using text file as we saw earlier, as it is much efficient and helping.

We will be using the famous database technology MySQL.

Task-1 - Download and install mySQL from their website.

Note - Python interpreter comes with support for working with databases, but nothing specific to MySQL. Python provides a database API (application programmer interface) for working with SQL-based databases known as DB-API. This removes the need to understand the details of database technologies.
* We just need a driver to connect DB-API to an actual database technology. (MySQL).

Task-2 - Install a MySQL Database driver for Python to connect with MySQL. by using link - https://dev.mysql.com/downloads/connector/python/

- install the driver using cmd in the newly created folder from Windows and write > py -3 setup.py install

Task-3 - Create our website's database and tables required for log. To do this, we are going to interact with MySQL server using its command-line tool which is a small utility that you start from our terminal window. (cmd)

To start the console - mysql -u root -p

Any commands you type at the console prompt are delivered to MySQL server for execution.

- To create a database for our webapp -

mysql > create database vsearchlogDB;
                        name

- To create a user ID and password specifically for our webapp -

mysql > grant all on vsearchlogDB.* to 'vsearch' identified by 'vsearchpasswd';
                                    username   password

- When you are done with the console, use the below command to quit. — mysql > quit.

Now that we have created the database, we need to create tables for within for our app.

- Before doing so, login as the Database's user using its password and ID

* The SQL statement we use to create required table is log.

Code mysql > create table log (
→ id int auto_increment primary key,
→ ts timestamp default current_timestamp,
→ phrase varchar (128) not null,
→ letters varchar (32) not null,
→ ip varchar (16) not null,
→ browser_string varchar (256) not null
→ results varchar (64) not null) ;

To view the table, issue. mysql > describe log ;

Note - We want our webapp to manually add statements in the table.
Hence, we need to write some python code to interact with
log table. For which, we need knowledge about Python's DB-API.

DB-API step 1: Define your connection characteristics.

```
>>> dbconfig = {'host': '127.0.0.1',
                'user': 'vsearch',
                'password': 'vsearchpasswd',
                'database': 'vsearchlogDB',}
```

The four pieces of information needed to connect with mySQL.

DB-API step 2: Import your database driver

```
>>> import mysql.connector
```

DB-API step 3: Establish a connection to the server.

```
>>> conn = mysql.connector.connect(**dbconfig)
```

*The "**" notation tells the connect function that a dictionary of arguments is being supplied in a single variable. On seeing, the "**" the connect function expands the dictionary arguments into four individual arguments which are used to connect to MySQL.

DB-API step 4: Open a cursor to send commands to the server, and to recieve results.

```
>>> cursor = conn.cursor()
```

Note - We are saving both the cursor and connection to variables so that we can call functions on them.

PB-API Step 5: Do the SQL thing!

As a general rule, we should use triple quoted strings, then assign it to a variable and then send the task to database server SQL. SQL Queries can be more than one line, hence triple quotes are useful. The below query shows the table of logs.

```
>>> _SQL = """ show tables """
>>> cursor.execute (_SQL)
```

- upon sending the query to SQL, they are not executed immediately; you have to ask for them. using three methods.

* Cursor.fetchone - retrieves a single row of results.
* Cursor.fetchmany - retrieves the number of rows specified
* Cursor.fetchall - retrieves all rows that make up results.

```
>>> res = cursor.fetchall ()
>>> res
[ ('log',) ] ← the table is called log.
```

To get more information about the table log -

```
>>> _SQL = """ describe log """
>>> cursor.execute (_SQL)
>>>> res = cursor.fetchall()
>>> for row in res:
        print (row)
```

Hepaximin

* To insert data into log table, we should use Placeholders instead of hard coding -

```
>>> SQL = """insert into log
            (phrase, letters, ip, browsen_string, results)
            values
            (%s, %s, %s, %s, %s) """
```

```
>>> cursor.execute(_SQL, ('hitch-hiker', 'xyz', '127.0.0.1',
                          'Safari', 'set()'))
```

Note *- We are using '()' in data values to store all five as single value, as cursor.execute() accepts at max two values.

Next step - use conn.commit to force the database to write the data.

```
>>> conn.commit()
>>> _SQL = """ select * from log """
>>> cursor.execute(_SQL)
>>> for row in cursor.fetchall():
        Print(row)
```

DB-API Step 6: Close your cursor and connection.

```
>>> cursor.close()
True
>>> conn.close()
```

Chapter 7 - final code

```
'import mysql.connector
def log_request (req: 'flask_request', res: str) -> None:
    """Log details of the web request and results."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    conn = mysql.connector.connect (**dbconfig)
    cursor = conn.cursor()

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s, %s)"""

    cursor.execute (_SQL, (req.form['phrase'],
                           req.form['letters'],
                           req.remote_addr,
                           req.user_agent.browser,
                           res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

Chapter - 8 - A little bit of Class

~~With~~ we need to create a class in order to hook into the with statement from previous chapter.

In this chapter we will learn about encapsulation. which is needed to create a context Manager.

A class ~~tetus~~ lets us bundle behavior and objects together in an object. which helps us jot down similar objects, ultimately saving tons of code-writing.

In Python we define a class behavior by creating a method. Method = class's function.
In Python, class variables are called attributes.

Object instantiation is when we create an object of the class's object.

Note - We can create empty classes with no attributes or functions.

```
>>> class CountFromBy:
        pass.
```

Self. Note - This technique helps us create abstract ~~methods~~ classes.

To create objects from a class use -

```
a = CountFromBy()
b = CountFromBy()
```

self. Note - When creating functions, do not use uppercase letters, and use "_" for space.
- When creating a class, use uppercase letters for first letter of each word with no space.

example — def count_from_by(): ← function
class CountFromBy(): ← class.

When you create an object from a class, each object shares the class's methods but maintains its own copy of attributes.

When we create an object, and assign it to a variable, it's value starts from 0, unless stated.

increase() increments the value by 1 unless stated.

```
>>> d = CountFromBy(100)
<= Coun >>> d
100
>>> d.increase(2)
>>> d
102
```

d.increase actually means CountFromBy.increase(c).

When creating a method, we should pass <u>self</u> as an argument as an alias to current object.

Self should always be the first argument, as invoking an object is matched with it.

When variables are defined within a function's suite, they exist while function runs. Basically, the variables are in "scope", both visible and usable within the function's suite only.

One way around it, is to assign the value to a variable so that it is not destroyed after the function finishes its work.

Basically, the rule is, if you need to refer to an attribute in your class, you must prefix the attribute name with self. Example - self.val

Use dunder 'init' to initialize the attribute values.

Another magic method is __repr__, which allows us to control how an object appears when displays at (>>>) prompt, as well as when print BIF

Final Code —

```
Class CountFromBy:

    def __init__(self, v: int = 0, i: int = 1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

Chapter-9 - The Context Management Protocol.

Protocol is an agreed procedure / set of rules that is to be adhered to.

To manage contexts with methods, our class must define at least two magic methods - __enter__ and __exit__. (This is the protocol).

The dunder enter can return a value to the with statement.

If you create a class that defines __enter__ and __exit__, the class is automatically regarded as a context manager by the interpreter, and can, as a consequence, hook into "with".

We can also add other methods to our context manager class as needed.

A good technique is to use dunder init before dunder enter. In doing so, we can initialize the connection to SQL server before, as well as keep things tidy.

To create our very own context manager, the context management protocol expects us to provide three things in order to hook into with statement -

1) An __init__ method to perform initialization (if needed)
2) An __enter__ method to do any setup
3) An __exit__ method to do any teardown (tidying-up).

We have to import mysql.connector before building our DBcm class.

DBcm.py

```
import mysql.connector
class UseDatabase:
    def __init__(self, config: dict) -> None:
        self.configuration = config

    def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

Chapter - 10 - Function Decorators

It is not a good idea to make the viewlog available to all users.

The web is stateless, given that it does not care what the request/result are, it only focuses on getting the ball and passing it ASAP.

Every web request is independant of ~~what~~ each other.

HTTP, Hyper Text Transfer protocal, states that all web should work like that. In doing so, higher performance is achieved at the expense of storing information.

Although variable usage sounds like a good idea, it is not feasible as using a variable would make it global, and eventually be lost/destroyed by the web once the code stops/re-runs.
Moreover, If there are two users, they both directs towards the same variable, and same boolean value, which won't work.
Example - logged_in = True
If __name__ == __main__:
app.run(debug=True).

In short, it is a bad idea to store our web app's state in global variables.

To overcome the problems from earlier, Most web app development frameworks provide "session". technology.

Session is a layer of state spread on top of the stateless web
- Like a cloth for picnic in a park for a family

By adding a small piece of identification data to our browser (a cookie), and linking this to a small piece of identification data on the web server, (Session ID), Flasks keeps everything in control.

To use Flask's session technology, we need to import session from flask module.

Cookies are formed for every connection we make to the webapp
For example, Mozilla, Chrome and Opera can be connected to the webapp at once but with three different connections. Thanks to a unique cookies assignment.

A decorator allows us to augment an existing function with extra code, and it does this by letting us change the behavior of the existing function without having to change its code.

We can not check a dictionary for a key's value until a key/value pair exists. Trying to do results in an key Error. Hence we do not use False Boolean to check logged in status as we should pop the value of 'logged_in' when logging out.

Functions can take another function as an argument.

Functions can be nested inside functions. (Remember Recursion, type-ish).

We can return a function from a function.

We can get our function to accept any number of arguments if we use (*args) when defining it.

```
def myFunc (*args):
    for a in args:
        print (a, end = ' ')
    if args:
        print()
```

Way-2

```
my func (1,2,3,4):
    fo. _ _ _ _
```

Note - Think of *args as a list of arguments. We have to define the args list first.

To pass a dictionary to a function, use (**dictionary). In doing so, the interpreter will break them and match their keys with values.

Note - Both these techniques can be mixed and matched.

```
def myfunc3 (*args, **kwargs):
    if args:
        for a in args: Print (a, end=' ')
    if kwargs:
        for k,v in kwargs.items():
            Print (k, v, sep='→', end = ' ')
    Print()
```

To

To create a decoration, you need to know that-

1) A decoration is a function.
2) A decoration takes the decorated function as an argument.
3) A decoration returns a new function.
4) A decoration maintains the decorated function's signature.

* After creating the decoration just call the decorating function before the function you want to use. Using @ syntax.

Basic code to create a decoration-

```
from functools import wraps !
def decoration_name (func) :
    @wraps (func)
    def wrapper (*args, **kwargs) :
        # 1. Code to execute BEFORE calling the decorated function.

        #2. Call the decorated function as required, returning its
            result if needed.
        return func (*args, **kwargs)

        #3. Code to execute INSTEAD of calling the decorated
            function.
return wrapper
```

~~Chap~~

Chapter 10's decoration -

```
from flask import session
from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are not logged in.'
    return wrapper
```

sneek peek of decoration use :

```
@app.route('/page')
@checked_logged_in
def page2() -> str:
    return 'This is page 2.'
```

Chapter - 11 - Exception Handling

When something goes wrong with our code, Python raises a runtime exception.

Using try: except syntax, we can sort as well as deal with errors.

If we know exactly what error is raised, we can use raise error except Error Name syntax, else just use except.

There can be more than one exceptions raised.

We should always try: to log what error occured, so that we can fix i-

~~sys exc_info~~

exc_info of the sys module provides information on the exception currently being handled. It returns the type, value and traceback object when invoked.

```
try:
    1/0
except:
    err = sys.exc_info()
    for e in err:
        print (e)
```

Result

```
<class 'ZeroDivisionError'>
division by Zero
<traceback object at 0x105b22188>
```

We can extend the 'except Exception' statement with the 'as' keyword to assign the error to a variable, and later on print it.

Example- except Exception as err:
    print ('some error has occured:', str(err))

Always try to handle errors silently.

When creating an error, pass "Exception" as an argument to the error class, so that it can inherit everything from "Exceptions" and you can simply "pass" the error class.

Chapter-12 - Advanced iteration

CSV (comma seperated value) files are used because they are very easy to import and work with. They are basically text-files which contains information.

We can use "with open ... as file" command to work with CSV files.

To output the CSV data as lists, use csv.reader(data) function.

To output the CSV data as dictionary, use csv.DictReader(data) function.

Python has a module to work with CSV files - csv - which has to be imported.

The down-side of CSV files is that it has all its data in uppercase letters.

strip() function removes whitespace, which makes our texts look good.

Strip method only works on strings and not lists, hence we should use ".split(',').strip()" to create a string out of a list and then strip it.

To convert Uppercase letters to Lowercase, use .title methods on strings.

To convert 24-hour time to 12-hour time, import datetime from datetime and create a function and call it.

```
from datetime import datetime.
def convert2ampm(time24:str) -> str:
    return datetime.strptime(time24,'%H:%M').strftime('I:%M %
```

Note.

To iterate over dictionaries, use "for k, v in flights.items()" syntax.

Comprehensions are faster and makes it fun to used instead of for-loops.

List-Comp

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())
```

→ Comprehension

```
more_dests = [dest.title() for dest in flights.values()]
```

It is worth it to learn comprehension, as it works in situations where for loops do not.

Dict-Comp

```
flights2 = {}
for k,v in flights.items():
    fltg flights2 [convert2ampm(k)] = v.title()
```

instead use -

~~more_flts~~

Comp-version → more_flights = {convert2ampm(k): v.title() for k,v in flights.items()}

Set comps and dict comps both use curly braces. To tell them apart, look for the ":" colon. If it is there, it is a dict-comp.

Tuple compresion cannot be made, as it does not make sense to create it out of an immutable data ~~type~~ structure.

When we use "()" in place of "[]", they are called generators. and they operate differently than list-comps.

Generators let go of data as soon as they are processed, whereas list-comps lets go of data all at once.

Generators are the best way to go, as they are fast and responsive. when working with huge amount of data.

When we call "return" in a function, it terminates. To overcome this, use "yield" which will carry on further tasks.