PROGRESS TRACKING

- CHAPTER 1 : PYTHON BASICS
    - 1.1 Text Strings :
    - 1.2. FILE IO
      - Reading Files
      - Writing Files
    - 1.3. Lists
        - Sample FILE IO Code
    - 1.4. Tuples
    - 1.5. Sets
    - 1.6. Dictionaries
    - 1.7 Exceptions
    - 1.8 Object and Classes
        - Inheritance
    - 1.9 Modules
    - 1.10 Script Writing, `␣␣name␣␣ == ␣␣main␣␣`
    - 1.11 Packages
    - Python: It Fits Your Brain
- CHAPTER 2 : OPERATORS, EXPRESSIONS, and DATA MANUPULATIONS
    - 1. Object Comparison :
    - 2. Ordered Comparison Operators :
    - 3. Conditional Expressions :
    - 4. Operations on Iterables :
    - 5. Operations on Sequences :
    - 6. Operations on Mutable Sequences :
    - 7. Operations on Sets
    - 8. Operations on Mappings
    - 9. List, Set & Dict Comprehensions
    - 10. Generator Expressions
- CHAPTER 3: Program Structure and Control Flow
    - 3.1 Loops & Iterations
    - 3.2 Exceptions
        - 3.2.1 Exception Hierarcy
        - 3.2.2 Exceptions and Control Flow

CHAPTER 1 : PYTHON BASICS

1. When using python in interactive mode : '_' variable holds the result of the last operation.
2. To end program use quit() or EOF, in macos/UNIX ctrl+D is EOF

> **It is common to use #! to specify the interpreter on the first line of a program**

```
#!/Users/ayushdutta/.pyenv/shims/python
```

- We can use a fstring inside a print() :

```
print(f'{year:>3d} {principal:0.2f}')
```

Here the :>3d means a 3 decimal number, and 0.2f means round of to 2 deciman accuracy.

> **NOTE : The round() function inplements "baker's rounding" .If the value being rounded is equally close to two multiples, it is rounded to the nearest even.**

> **NOTE : Python does not have ++ or -- operator.**

Walrus Operator : Sometimes, you may see the assignment of a variable and a conditional combined together using the := operator.

```
x = 0
while((x := x+1) < 10):
    print(x)                # prints 1, 2, 3, 4, 5, ... , 9
```

- The `break` statement just breaks the innermost loop.
- The `continue` statement skips the rest of the loop body.
- `''' xyz '''` is multiline comment, which captures all string in between

1.1 Text Strings :

- `fstrings**` :

```
base_year = 2020,
print(f'{base_year + year:>4d} {principal:0.2f}')
```

- `format()` and `%` operators are also used alternative to fstring

```
print('{0:>3d} {1:0.2f}'.format(year, principal))
print('%3d %0.2f' % (year, principal))
```

- for strings, `string[-1]` refers to the last index
- **String Slicing** : `string[1:5]` from 1st index to 4th.
- `.replace('hello', 'hello2')` function replaces all the occurance.
- **Common String Methods :**

Method	Description
`s.endswith(prefix[,start [,end]])`	string ends with prefix
`s.find(sub [, start[,end]])`	Finds the first occurrence of the specified substring sub or -1 if not found.
`s.replace(old, new[,maxreplace])`	Replaces a substring, n times
`s.split([sep[,maxsplit]])`	splits string into maxsplit number of splits
`s.strip([chrs])`	Removes whitespaces but removes chars if chrs is given

- **Non-string values can be converted into a string representation by using the str(), repr(), or format() functions.**

```
s = 'hello\nworld'
str(s)
repr(s)
x = 10.235
format(x, "0.2f")
f'{x:0.2f}'
```

output :

```
hello
world
hello\nworld
"10.24"
"10.24"
```

1.2. FILE IO

open(FILE) returns a `file oject`

Reading Files

```
with open('data.txt') as file:
  for line in file:
    print(line, end='')
file.close()
```

The `with` statement opens the object as file. file only valid in context.

```
with open('data.txt') as file:
  data = file.read()
```

Reads the whole file at once.

```
with open('data.txt', encoding='utf-8') as file:
  while (chunk := file.read(10000)):
    print(chunk, end='')
```

Reads file in chunks.

Writing Files

```
with open('data.txt', 'wt') as fp:
  while year <= numyears:
    principal = principal * (1+rate)
    print(f'{year:3d} {principal:0.2f}', file=fp)
    year += 1
```

OR

```
fp.write(f'{year:3d} {principal:0.2f}\n')  #NOTE the \n
```

Input from Console

```
name = input("What is your name")
print("hello", name)
```

1.3. Lists

```
names = ['Dave', 'Paula', 'Thomas', 'Lewis']
last_index = names[-1]

# splits the letters into a list
letters = list('Dave')

# Multi Dimentional list
a = [1, 'Dave', 3.14, ['Mark', 7, 9, [100, 101]], 10]
value_100 = a[3][3][0]
```

Sample FILE IO Code

```
# TODO
# Fileformat : Name,QTY,PRICE
# 1. read filename from argv
# 2. Find total price  accross all rows

import sys

if len(sys.argv) != 2:
    raise SystemExit(f'Usage: {sys.argv[0]} filename')

rows = []

with open(sys.argv[1], "rt") as file:
    for line in file:
        rows.append(line.split(',')[1:])

total = sum([int(row[0]) * float(row[1])  for row in rows])

print(f"SUM : {total}")
```

1.4. Tuples

`immutable`

To create simple data structures, you can pack a collection of values into an immutable object known as a tuple

Tuples can do indexing, slicing, concat, etc but they can not be modified once created.

```
# File containing lines of the form "name,shares,price"

filename = 'portfolio.py.txt'

portfolio = []

with open(filename) as file:
    for line in file:
        row = line.split(',')
        name = row[0]
        shares = int(row[1])
        price = float(row[2])

        holding = (name, shares, price)
        portfolio.append(holding)

# iterating over each holding
total = 0
for _, shares, price in portfolio:
    total += shares * price;

print(f"Total portfolio : {total}")
```

1.5. Sets

`immutable`

The elements of a set are typically restricted to **immutable** objects. For example, you can make a set of numbers, strings, or tuples.

> **Unlike lists and tuples, sets are unordered and cannot be indexed by numbers. They must have unique values**

```
print(portfolio)
s = { sets[0] for sets in portfolio }
```

```
[('GOOG', 100, 490.0), ('FB', 88, 542.0), ('ALPH', 33, 23.0), ('GOG', 18, 45.0)]
{'FB', 'ALPH', 'GOOG', 'GOG'}
```

Operations	Explanation
a = t | s	Union
b = t & s	Intersection
c = t - s	Difference
d = t ^ s	Symmetric Difference

Symmetric Difference - items that are in either s or t but not in both.

Operation	Explanation
a.add('Hello')	Single item add
a.update('hello', 'hi', 'yo')	Add multiple items
t.remove('IBM')	Remove 'IBM' or raise KeyError if absent.
s.discard('SCOX')	Remove 'SCOX' if it exists.

1.6. Dictionaries

ref code = dicts.py

`d = dict()` #empty dict

```
s = {
    'name' : 'GOOG',
    'shares' : 100,
    'price' : 490.10,
    'S&P500' : False
  }

  print(s['names])
    if s.get('S&P500') is False:
       s['S&P500'] = True

  print(s)
```

```
'GOOG'
{'name' : 'GOOG', 'shares' : 100, 'price' : 490.10 'S&P200' : true}
```

Using tuples with dicts

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42     # Parens omitted

> {('IBM', '2015-02-04'): 91.42}
```

> **NOTE - Any kind of object can be placed into a dictionary, including other dictionaries. However, mutable data structures such as lists, sets, and dictionaries cannot be used as keys.**

Example Dict Code

```
portfolio = [
   ('ACME', 50, 92.34),
   ('IBM', 75, 102.25),
   ('PHP', 40, 74.50),
   ('IBM', 50, 124.75)
]
total_shares = { s[0]: 0 for s in portfolio }

for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

We can do the same thing without initializing the total_share with zeros and keys. We use the `counter() function`
Basically counter generates a Counter() object with `Keys:number` items.

```
from collections import Counter()

total_shares = Counter()

for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_share = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

Common Functions

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict()  # -> creats a dicts
print(list(d))  # -> creats a list of 'keys'
d.pairs() # -> creates a “keys view” object that is attached to the dict
```

1.7 Exceptions

We use `try` `except` to handle errors

```
try :
  ...
except ValueError as err:
  ...
```

also

```
raise RuntimeError('Computer says no')
```

```
raise SystemExit("EXIT") # Exit program with no error
```

1.8 Object and Classes

All values used in a program are objects. An object consists of internal data and methods that perform various kinds of operations involving that data.

The `class` statement is used to define new types of objects and for object-oriented programming.

ref : class_test.py

```
class Stack:

    def __init__(self) -> None:
        self._items = []

    def push(self, item):
        self._items.append(item)

    def pop(self):
        self._items.pop()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return f'<{type(self).__name__} at 0x{id(self):x}, size={len(self)}>'

    def print(self):
        print(self._items)

s = Stack()
s.push('ONE')
s.pop()
for item in ['ONE', 'TWO']:
    s.push(item)
s.push('THREE')
s.print()

print(f'len = {len(s)}')
print(s)
```

`self` -> The term “self” refers to the instance of the class that is currently being used. Its always the first argument. **Pointer to Current Object.**

`__init__(self, ...)` -> Its the class constructor. Gets called when instance of class is created. It can be used to initialize the attributes.

`_method` are meant to be private methods. Its a convention. Not to be used outside the Class.

`__method__` are special method. eg :. `__len__` gets called when `len(obj)` is called.
`__repr__` changes the way how the object is displayed and printed.

Inheritance

Now we can create a Class that inherits all attributes and methods of another class.

```
class MyStack(Stack):
    def print_len(self):
        return "len : " + str(len(self._items))

s1 = MyStack()
s1.push('ONE')  # -> calling push() from Stack class
s1.push('TWO')  # -> calling push() from Stack class
print(s1.print_len())  # -> calling from MyStack class
s1.print()
```

We can also change the behaviour of an existing method in the inherited class. For ex: we can make the stack to accept only numbers.

```
class MyStack(Stack):
    def print_len(self):
        return "len : " + str(len(self._items))

    ## CHANGING METHOD behaviour of Parent class
    def push(self, item):
        if not isinstance(item, (int, float)):
            raise TypeError('Expected an integer value')
        super().push(item)

s1 = MyStack()
# s1.push('ONE') -> ERROR
s1.push(1)
print(s1.print_len())
s1.print()

#  len : 1
# [1]
```

1.9 Modules

As your programs grow in size, you will want to break them into multiple files for easier maintenance. To do this, use the import statement. To create a module, put the relevant statements and definitions into a file with a .py suffix and the same name as the module.

check : portfolio_package/portfolio_value_calculator.py

1.10 Script Writing, `__name__ == __main__`

`__name__ = name of program` : if imported as module
`__name__ = __main__` : if ran as standalone program

now we can use `if __name__ == __main__ :` to make the program run standalone in a proper way.

check : portfolio_package/readport.py

1.11 Packages

```
portfolio_package/
    __init__.py
    readport.py
    portfolio_value_calculator.py    (imports readport.py)
```

To run the portfolio_value_calculator.py, we have to go out of this package and run it :
`python -m portfolio_package.portfolio_value_calculator`

Python: It Fits Your Brain

CHAPTER 2 : OPERATORS, EXPRESSIONS, and DATA MANUPULATIONS

Number	Base
42	Decimal
0b101010	Binary Number
0o52	Octal Number
0x2a	Hexadecimal

> Base is not stores as part of the integer. All of the above literals will display as 42. To print number with base use `bin(x), oct(x), hex(x)` .

- `Floating Point number` - 4.2e+2 : Stored as **IEEE 754 double- precision (64-bit) values**.
- We can't include the assignment operator as part of an expression:

```
while line=file.readline(): # Syntax Error.
    print(line)

while (line:=file.readline()): # Using the wallrus operator
    print(line)
```

- Reference to lists (pointers in python). Both are refering to the same list at same memory location.

```
a = [1, 2, 3]
b = a
a += [4, 5]

print(a)    # -> [1, 2, 3, 4, 5]
print(b)    # -> [1, 2, 3, 4, 5]
```

1. Object Comparison :

`x is y` checks if `id(x) == id(y)`. Test two values to see whether they refer to literally the same object in memory. `x == y` checks if the containts of the objects are equal and not the object itself.

2. Ordered Comparison Operators :

We can use the comaprison operators on lists, tuples, and strings. It compares each index to the corresponding index. As soon as the furst one greater or smaller is encountered, it decides the total outcome.

- `[1, 2, 3]` < `[3, 4, 5]` also `[3, 4, 5, 6, 9, 9]` < `[1, 2, 4]`
  For sets, x < y tests if x is strict subset of y (i.e., has fewer elements, but is not equal to y).
- For sets it checks strict subsets as sets values are unique.

3. Conditional Expressions :

```
minvalue = a if a <= b else b
```

4. Operations on Iterables :

Description	Operations
`for vars in s:`	Iteration
`a, b, c ... in s:`	Variable unpacking
`x in s`, `x not in s`	Membership
`[a, *s, b], {a, *s, b}`	Expansion in list, tuples or sets

Example 1: Unpacking list and creating dict

```
items = [3, 4, 5]
d={ }
d['x'], d['y'], d['z'] = items

# {'x' : 3, 'y' : 4, 'z' : 5}
```

> **NOTE : When unpacking values into locations, the number of locations on the left must exactly match the number of items in the iterable on the right.**

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, day, year), (hour, minute, am_pm) = datetime
(month, _, year), (hour, _, am_pm) = datetime    # _ is throwaway variable
```

Example 2 : If the number of items being unpacked isn't known use `*extras`

```
items = [1, 2, 3, 4]
a, *extras, b = items    # a=1, *extras=[2, 3], b=4

datetime = ((5, 19, 2008), (10, 30, "am"))
(month, *_), (hour, *_) = daytime
```

Any iterable can be expanded when writing out list, tuple, and set literals usinf star (*) operator.

```
items = [1, 2, 3, 4]
a = (10, 11, *items, 12)  # (10, 11, 1, 2, 3, 4, 12)
b = [a, *items, b] # [a, 1, 2, 3, 4, b] and not [a, [1, 2, 3, 4], b]
```

> NOTE : However, many iterable objects (such as files or generators) only support one-time iteration. If you use *-expansion, the contents will be consumed and the iterable won't produce any more values on subsequent iterations.

5. Operations on Sequences :

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

a[2:5]     # [2, 3, 4]
a[:3]      # [0, 1, 2]
a[-3:]     # [7, 8, 9]
a[::2]     # [0, 2, 4, 6, 8]
a[::-2]    # [9, 7, 5, 3, 1] , if stride is -ve initial is from end if i is omited
a[0:5:2]   # [0, 2, 4]
a[5:0:-2]  # [5, 3 ,1]
a[:5:1]    # [0, 1, 2, 3, 4]
a[:5:-1]   # [9, 8, 7, 6]
a[5::1]    # [5, 6, 7, 8, 9]
a[5::-1]   # [5, 4, 3, 2, 1, 0]
a[5:0:-1]  # [5, 4, 3, 2, 1]
```

Slices can be named using `slice()`. For example:

```
firstfive = slice(0, 5)
s = 'hello world'
print(s[firstfive])     # Prints 'hello'
```

6. Operations on Mutable Sequences :

All the same as above and insertion and deletion also allowed.

```
a = [1, 2, 3, 4, 5]
a[3:4] = [-1, -2, -3]    # a = [1, 6, 10, -1, -2, -3, 5]
a[2:] = [0]     # a = [1, 6, 0]

a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]      # a = [1, 10, 3, 11, 5]
# Have to suply exact number of items in the right that matches the output on the left ca
del a[1:]  # [1]
```

> NOTE : The popular numpy package has different slicing semantics than Python lists. Same goes with third party packages.

7. Operations on Sets

Operations	Explaination
a = t | s	Union
b = t & s	Intersection
c = t - s	Difference
d = t ^ s	Symetric Difference

Symetric Difference - items that are in either s or t but not in both.

> Also works on keys of dicts.
> eg : `a.keys() & b.keys()`

8. Operations on Mappings

Operation	Description
`a = m['key']`	Inexing
`del m['key']`	Deleting by key
'key' in m	Membership
`m.keys()`	Returns the Keys
`m.values()`	Returns the Values
`m.items()`	returns (key, value) pair

example : tuple as key (immutble)

```
m[('ayush', 'dutta', '11-07-2002')] = 'Computer Science'
m[('lugen marie','dutta', '11-28-2003')] = 'English'
```

9. List, Set & Dict Comprehensions

Find all entries that have more than 100 shares

```
# Some data (a list of dictionaries)
portfolio = [
  {'name': 'IBM', 'shares': 100, 'price': 91.1 },
  {'name': 'MSFT', 'shares': 50, 'price': 45.67 },
  {'name': 'HPE', 'shares': 75, 'price': 34.51 },
  {'name': 'CAT', 'shares': 60, 'price': 67.89 },
  {'name': 'IBM', 'shares': 200, 'price': 95.25 }
]

more100 = [s["name"] for s in portfolio if s["shares"] > 100]

# Use of walrus operator, and creating a set of tuples
more100_set = {(s["name"], v) for s in portfolio if (v := s["shares"]) >= 100}
print(more100_set)
```

> NOTE : typechecking while comprehension does not exist. So use seperate function. eg : comprehension.py

10. Generator Expressions

A generator expression is an object that carries out the same computation as a list comprehension but produces the result iteratively. It gives you a generator object that we can iterate on demand.

```
nums = [1, 4, 9, 16]
squares = (i*i for i in values)

>>> squares
<generator object at 0x590a8>
>>> next(squares)
1
>>> next(squares)
4
...
>>> for n in squares:
...     print(n)
9
16
>>>
```

> **NOTE : A generator expression can only be used once. If you try to iterate a second time, you'll get nothing.**

Look at generators.py for implementation using FILE IO.

CHAPTER 3: Program Structure and Control Flow

3.1 Loops & Iterations

1. `enumerate` -

```python
values = [1, 2, 3, 4, 5]

for index, i in enumerate(values, start=100):
    print((index, i))

# (100, 1)
# (101, 2)
# (102, 3)
# (103, 4)
# (104, 5)
```

2. `zip` -

```python
x = [1, 2, 3, 4, 5]
y = ["a", "b", "c", "d", "e"]

for i in (zip_x_y := zip(x, y)):
    print(i)

# (1, 'a')
# (2, 'b')
# (3, 'c')
# (4, 'd')
# (5, 'e')
```

3. `for else` loop structure

The else runs only if the loop is complleted on its own without breaking at any moment.

```
values = [1, 2, 3, 4, 6, 7, 8, 9, 6, 12, 18, 19]

for i in values:
    if i % 5 == 0:
        print("Number div by 5 found")
        break

else:
    print("No number div by 5")
```

> **NOTE : to break out of a deeply nested loop raise an exception**

3.2 Exceptions

check : exceptions.py

As a matter of programming style, you should only catch exceptions from which your code can actually recover.

Exceptions can be use with `as` var. And one can use the `a.args` which is a tuple

```
try:
    file = open("foo.txt", "r", encoding="UTF-8")

except FileNotFoundError as e:
    print(e.args)

# (2, 'No such file or directory')
```

It also supports `try else` statements. The Else executes if there is no exception raised in try block.

```
try:
    file = open("./file.txt", "r", encoding="UTF-8")

except FileNotFoundError as e:
    print(f"Unable to open file : {e}")
    DATA = ""

else:
    DATA = file.read()
    print(DATA)
```

3.2.1 Exception Hierarcy

For `IndexError` or `KeyError` we can use `LookupError` ro handle both. Both of them inherit from `LookupError`.

Exception Catagories :

Exception Class	Description
`BaseException`	The root class for all exceptions
`Exception`	Base class for all program-related errors
`ArithmeticError`	Base class for all math-related errors
`ImportError`	Base class for import-related errors
`LookupError`	Base class for all container lookup errors
`OSError`	Base class for all system-related errors
`ValueError`	Base class for value-related errors, including Unicode
`UnicodeError`	Base class for a Unicode string encoding-related errors

Other exceptions which aren't part of a larger exception group :

Exception Class	Description
`AssertionError`	Failed assert statement

Exception Class	Description
`AttributeError`	Bad attribute lookup on an object
`EOFError`	End of File
`MemoryError`	Recoverable out-of-memory error
`NameError`	Name not found in the local or global namespace
`NotImplementedError`	Unimplemented feature
`RuntimeError`	A generic “something bad happened” error
`TypeError`	Operation applied to an object of the wrong type
`UnboundLocalError`	Usage of a local variable before a value is assigned

3.2.2 Exceptions and Control Flow

Few exceptions can alter the control flow.

`SystemExit` - Raised to indicate program exit. The message is printed to `sys.stderr`