Python

Operation

```
a += 2
a -= 2
a *= 2
a **= 2
a /= 2
a //= 2
a %= 2
```

Variables

```
age = input("How old are you)
```

String methods

```
len(`string`) // Len
ord('L) // Code point
chr(76) // code point char
"test".index("t) // Provided char index
"test".count("t") // sum of occurence of a char
list("test) // ToList
"test".endswith("!")
"test".startswith("t")
"test".find('x')
"test".isalnum() // Only alpha numeric
"test".isalpha() // Only alpha
"test".isdigit() // only digit
"test".islower() // all char lower case
"test".isupper() // all char upper case
".".join("x","y") // Single string based on delimiter
"xxxx".split()
"x b c".find('b') // Index of first occurance
"e f g".rfind('o') // Index of last occurance
f"Hello {name}! You are {age} years old" // String format
f"Balance: ${balance:.2f}"
```

Control

Conditions

```
if True:
   print("True")
else:
   print("False")
```

Type conversion

```
int("10")
float("10.3")
str(1.2)
set([1,2,3])
list("abc")
```

Loop

```
 for i in range(10):
```

Data structure

1D List

```
list_data = [] // Create
len(list_data) // Size
list_data.append(1) // Add
list_data.remove(1) // Remove
del list_date[1]
list_data[:1] , list_data[1:3]// slice
list_data.sort(), //Sort
list_data.sort(reverse=True) // Reverse Sort
max(list_data) // Max element
list_1 + list_2 // Concate
```

List Iteration

```
for value in sequence:
    print(value * 2)

for i, value in enumerate(sequence):
    print(f'{i}: {value}')
```

List Sorting

```
s = sorted(s, key = lambda x: (x[1], x[2])) //multi key
```

List Filtering

```
filter(lambda x: x > 1, nums_list)
evens = [i for i in range(10) if i % 2 == 0]
```

2D List

```
rows, cols = (5, 5)  //Create
arr = [[0]*cols]*rows
arr[1][4] = row 1, col 4.

for i in range(len(x)):
    for j in range(len(x[i])):
        print(x[i][j])
```

Linked List

```
def Node:
    def __init__(self,v)
        self.v = v
        self.next = None

Define head
```

Hash

```
D = {
    "a": 1,
    "b": 2,
    "c": 3

}
D["a"] // Access
len(D) // Length
D["s"] = 2 // Add
del D["s"] // Remove
for x in D.keys(): // loop on keys
```

```
    print(x)
for x in D.values(): // loop on values
    print(x)
for k, v in canadian_capitals.items():
    print(k)
D.setdefault("s",2)
D.get("s")
```

Default Dict

```
from collections import defaultdict

word_counts = defaultdict(int)
```

Counter

```
from collections import Counter
a = Counter([1, 1, 3, 2, 1, 3, 4, 1])
d.most_common(2) // Most common elenments
```

Dequeue

Stack

```
from collections import deque
#add/remove from the end of the list/deque.
a= deqeue()
a.append(4)
a.pop()
```

Queue

```
from collections import deque
q = dequeue()
q.appendleft(val)
q.pop()
```

OO

```
class Student:
    def __init__(self, name, grade, age):
```

```
        self.name = name
        self.grade = grade
        self.age = age
    def __repr__(self):
        return repr((self.name, self.grade, self.age))

student1 = Student("1","A",10)
student2 = Student("1","A",10)
x = [student1,student2]
sorted(x,key = lambda student: student.age)
```

Set

```
student_ids = set()
student_ids.add(123) // Add
456 in student_ids // Check
students_ids.remove(123) // Remove
student_set = set(student_list) // Convert
for x in student_set: // Iterate
    print(x)

intersaction = set1 & set2 // Intersaction
diff = set1 - set2         // diff
```

DS

Stack

```
S = []
S.append('s') // Push
S.pop() // Pop
```

Queue

```
S = []
S.append('a') // Enqueue
s.pop(0) // dequeue
```

Heap

```
import heapq
customers = []
heapq.heappush(customers,(2,"Harry"))
```

```
heapq.heappush(customers,(3,"Charles"))
while customers:
    print(heapq.heappop(customers))
```

Exception

```
  raise MethodNotAllowed('Bike has already been sold')
  raise ValueError()

try:
 name = "x"
 raise ValueError("test") // raise an exception
except ValueError as e:
 print("x")
finally:
 print("y")
```

More OO

```
class Stack:
  max_size = 0 // static attribute.

  @classmethod  // static method
  def max_size(cls):
    return cls.max_size

  def __init__(self):
     self.__stack_list = []

  def push(self,val):
     self.__stack_list.append(val)

class AddingStack(Stack):
   def __init__(self):
      super().__init__()
```

Enum

```
from enum import Enum

class Condition(Enum):
    NEW = 0
    GOOD = 1
```

```
    OKAY = 2
    BAD = 3
```

Data class

```
from dataclasses import dataclass

@dataclass(Order=True)
class Point:
    x: int
    y: int

    @property
    def x(self):
        return x
```

Dunder method

```
def __add__(self,other):
    if isinstance(other,timedelta):
        return timedelta(self._days + other._days)
```

Files

```
f = open(name,mode) // r / w / a / r+)
for line in file:
    print (each)
```

```
with open("geeks.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

```
with open("geeks.txt", "r") as file:
    data = file.read() // read the whole file
```

File action

```
os.rename(filename,new_filename) // Rename
os.remove(filename) // Delete
```

OS

```
import os
os.mkdir("my_new_directory")
os.rmdir("my_new_directory")
os.makedirs("a/b")
os.listdir()  // List dirs
```

Json

```
import json
x =  '{ "name":"John", "age":30, "city":"New York"}'
y = json.loads(x)
print(y["age"])
```

```
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

y = json.dumps(x)
print(y)
```

CSV

```
import csv

with open('data/read_example.csv', 'r') as file:
    reader = csv.DictReader(file)

    for person in reader:
        print(f'{person["Name"]} is {person["Age"]}')
```

Dates

```
import datetime
x = datetime.datetime.now()
y = datetime.datetime(2020,5,17) // create
print(time.hour)
print(time.minute)
print(time.second)
print(time.microsecond)

now = datetime.datetime.now()
print(now)
print(now.date())
print(now.time())
```

Tree in python

Binary search tree Node

```

```

Binary search tree search

Binary search tree traverse

```
def visit(node):
    if not node:
        return
    visit(node.left)
    print(node.value)
    visit(node.right)
```

Preorder

```
def visit(node):
    if not node:
        return
    print(node.value)
    visit(node.left)
    visit(node.right)
```

Graphs in python

Dundurf methods

- **eq**(self, other): Defines behavior for the equality operator, ==.
- **ne**(self, other): Defines behavior for the inequality operator, !=.
- **lt**(self, other): Defines behavior for the less-than operator, <.
- **gt**(self, other): Defines behavior for the greater-than operator, >.
- **le**(self, other): Defines behavior for the less-than-or-equal-to operator, <=.
- **ge**(self, other): Defines behavior for the greater-than-or-equal-to operator, >=.