Integer Truncation

The `round` function uses "round half to even" strategy, also known as "bankers' rounding" or "unbiased rounding." When a number is exactly halfway between two possible rounded values, it rounds to the nearest even number.

```
import math

print("ROUND")
print(round(3)) # 3
print(round(3.49)) # 3
print(round(3.5)) # 4
print(round(2.5)) # 2

print("CEIL (ROUND UP)")
print(math.ceil(3)) # 3
print(math.ceil(3.49)) # 4
print(math.ceil(3.5)) # 4

print("FLOOR (ROUND DOWN)")
print(math.floor(3)) # 3
print(math.floor(3.49)) # 3
print(math.floor(3.5)) # 3
```

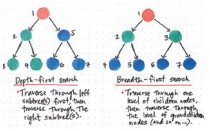
Comparing Strings

In Python, when comparing strings, the comparison is done `lexicographically` (i.e, dictionary order). The comparison is performed character by character from left to right.

```
print('abc' > 'abbbbbbb') # True
```

1. At the first position, `'a'` is the same in both strings.
2. At the second position, `'b'` is the same in both strings.
3. At the third position, `'c'` is greater than the corresponding character `'b'` in the second string.

```
print('123' > '111111111') # True
```



BFS (Breadth-First Search)

Big Fat Shit is an algorithm used to traverse or search tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and **explores the neighbor nodes at the present depth before moving on to nodes at the next depth level.**

```
from collections import deque

# Breadth-First Search (BFS) Algorithm
def bfs(graph, start):
    # Initialize an empty set to track visited nodes
    visited = set()
    # Initialize a deque (double-ended queue) with the starting node
    queue = deque([start])
    # Continue until the queue is empty
    while queue:
        # Dequeue the node at the front of the queue
        node = queue.popleft()
        # Check if the node has not been visited
        if node not in visited:
            # Mark the node as visited
            visited.add(node)
            # Enqueue unvisited neighbors for further exploration
            queue.extend(graph[node] - visited)
    # Return the set of visited nodes
    return visited

graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

start_node = 'A'
bfs_result = bfs(graph, start_node)
print(bfs_result)
```

```
{'G', 'F', 'C', 'D', 'E', 'H', 'B', 'A'}
```

DFS (Depth-First Search)

Dip Fucking Shit is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph) and **explores as far as possible along each branch before backtracking.**

```
# Depth-First Search (DFS) Algorithm
def dfs(graph, node, visited=None):
    # Initialize a set for visited nodes if not provided
    if visited is None:
        visited = set()
    # Mark the current node as visited
    visited.add(node)
    # Explore unvisited neighbors using recursion
    for neighbor in graph[node] - visited:
        dfs(graph, neighbor, visited)
    # Return the set of visited nodes
    return visited

start_node = 'A'
dfs_result = dfs(graph, start_node)
print(dfs_result)
```

```
{'G', 'F', 'D', 'C', 'E', 'H', 'B', 'A'}
```

- Three ways to use `range()` function:
  - `range(stop)` takes **one** argument:

```
for i in range(5):
    print(i)
```

  - `range(start, stop)` takes **two** arguments:

```
for i in range(3, 10):
    print(i)
```

  - `range(start, stop, step)` takes **three** arguments:

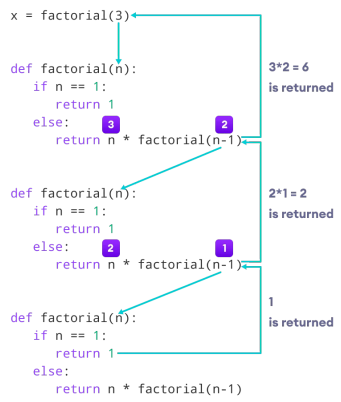
```
for i in range(3, 10, 4):
    print(i)
```

```
for i in range(5, 0, -1):
    print(i)
```

```
def super_fib_R(term2, n):
    if n == 1:
        return [1]
    elif n == 2:
        return [1, term2]
    else:
        result = super_fib_R(term2, n-1)
        result.append(sum(result))
        return result
```

```
def super_fib_I(term2, upper):
    result = [1, term2]
    total = sum(result)
    while total <= upper:
        result.append(total)
        total *= 2
    return result
```

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off



DICT

Function	Description
pop()	Remove the item with the specified key.
update()	Add or change dictionary items.
clear()	Remove all the items from the dictionary.
keys()	Returns all the dictionary's keys.
values()	Returns all the dictionary's values.
get()	Returns the value of the specified key.
popitem()	Returns the last inserted key and value as a tuple.
copy()	Returns a copy of the dictionary.

```
{ key : value, key : value }
country_capitals = {
  "United States": "Washington D.C.",
  "Italy": "Rome",
  "England": "London"
}

print(country_capitals["United States"])
 # Washington D.C.

country_capitals["Germany"] = "Berlin
 # add an item with "Germany" as key
and "Berlin" as its value

del country_capitals["United States"]
 # delete item having "United States" key
```

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

LIST

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

FILE+IO

With open('textfile.txt', 'w') as f: #open txt file with open() called f
    f.write('123')

Mode	Description
r	Open a file for reading. (default)
w	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Open a file for exclusive creation. If the file already exists, the operation fails.
a	Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Open in text mode. (default)
b	Open in binary mode.
+	Open a file for updating (reading and writing)

Tupling and Listing

When you cast a dictionary to a list or tuple it will only return the list/tuple of **only** the keys

```
print(tuple({2:4,5:6,7:8}))
print(list({'A':8,"C":"6",5:4}))
```

```
(2, 5, 7)
['A', 'C', 5]
```

- open() - takes two parameters; filename, and mode.
- write() - Writes the specified string to the file
- writelines() - writes a list of strings to the file
- close() - closes the file
- read () - returns the file content
- readable() - returns whether the file stream can be read of not
- readline() - returns one line from the file
- readlines() - returns a list of lines from the file

```
f = open("demofile3.txt", "w")
f.write("CS1010E CHEAT SHEET")
f.close()
```

True, False Statements

True = 1 [E.g., (True + 1) * 5 = 10]
False = 0 [E.g., (False + 3) * 5 = 15] OR [3**False = 1]

"and" statements: False if both False, True if both True, will run through both sides of the code
"or" statements: For z = x or y, if x is True, z will run even if y is an error, as it is already processed True on x, will only be False if both statement is False

String habits

1) **String can be added or multiplied**:
"ab" + "cd" = "abcd" OR "ab" * 2 = "abab"

2) **Order High or Low:**
"banana" < "bananb" (run through every letter)
"c" >= "banana" (length doesn't matter, first letter c > b already)
"banana" >= "banan" (even if uncompleted it is still <=)
*Note that > and >= in this case is the same

LIST Comprehension

E.g.,
C2 = [i for i in range(1, 101) if i not in c_list]
Means
for i in range(1, 101):
    If i not in c_list:
        C2.append(i)

Type	Notation	Example
Addition	+	5 + 6 = 11
Subtraction	-	5 – 6 = -1
Multiplication	*	5 * 6 = 30
Division (float)	/	5 / 6 = 8.3334
Modulus (Remainder after division)	%	For x % y, if x < y, = x 5 % 6 = 5 6 % 5 = 1 Find even numbers: i % 2 == 0
Exponential	**	5 ** 6 = 15625
Floor Division	//	For x // y, if x < y, = 0 5 // 6 = 0 6 // 5 = 1 (always round down to nearest int)

```
lambda arguments : expression
```

```
>>> (lambda x: x + 1)(2)
3
```

Indexing

Using same sequence, s:
s[x], x cannot be > len(s), even if negative x, also cannot go negatively out of range

SETS AND DICTIONARY

Sets: unordered collection with NO duplicate element
Unordered means: no indexing since there is no order

```
lst = [1, 2, 3]
# append(): Adds an element at the end of the list.
lst.append(4)  # [1, 2, 3, 4]
# extend(): Adds elements of a list to the end of the current list.
lst.extend([5, 6])  # [1, 2, 3, 4, 5, 6]
# insert(): Adds an element at a specified position.
lst.insert(1, 'a')  # [1, 'a', 2, 3, 4, 5, 6]
# remove(): Removes the first occurrence of the element with the specified value.
lst.remove('a')  # [1, 2, 3, 4, 5, 6]
# pop(): Removes the element at the specified position, or the last item.
lst.pop()  # [1, 2, 3, 4, 5]
# clear(): Removes all the elements from the list.
lst.clear()  # []
# reverse(): Reverses the order of the list.
lst.reverse()  # [5, 4, 3, 2, 1]
len(lst)  # Output: 5
min(lst)  # Output: 1
max(lst)  # Output: 5
print(1 in lst)  # Output: True
print(6 not in lst)  # Output: True
```

Sequences, List and Tuples:

Sequence are list, tuples and strings: all can + and *
min("snakes are dumb") = " "
max("snakes are dumb") = "s"

ASCII

ord('A') -> 65 ord('a') -> 97
chr(65) -> 'A' chr(97) -> 'a'

ord('Z') -> 65 ord('z') -> 122
chr(90) -> 'A' chr(122) -> 'z'

```
tup = (1, 2, 3, 2, 2, 3)
# count(): Returns the number of times a specified value occurs in a tuple.
tup.count(2)  # Output: 3
# index(): Searches the tuple for a specified value and returns the position.
tup.index(3)  # Output: 2
```

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# slicing [start:stop:step] (stop is exclusive)
# (start defaults to 0) (step defaults to 1)
# if start > stop or start > len(), empty list is returned
lst[1:4]  # Output: [2, 3, 4]
lst[3:]  # Output: [4, 5, 6, 7, 8, 9, 10]
lst[:3]  # Output: [1, 2, 3]
lst[::2]  # Output: [1, 3, 5, 7, 9]
lst[1:5:2]  # Output: [2, 4]
lst[0:6:-1]  # Output: []
lst[::-1]  # Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
s1 = '1234567890'
print(s1[8:2:-12])  # 9
print(s1[-2:2:-2])  # 975
print(s1[8:-7:-2])  # 975
print(s1[8:3:-2])  # 975
print(s1[-2:3:-2])  # 975
print(s1[8:4:-2])  # 97
```

Precedence

Precedence Level	Operators	Explanation
1 (highest / comes first)	()	Parentheses
2	**	Exponent
3	+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
4	*, /, //, %	Multiplication, Division, Floor division, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise shift operators
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	(*) is, is not, in, not in, ==, !=, >, >=, <, <=	Comparisons, Identity, Membership operators
11	not	Logical NOT
12	and	Logical AND
13 (lowest, comes last)	or	Logical OR

(*) The is, is not, in, not in have a higher precedence than the ==, !=, >, >=, <, <=

Lambda
A pseudo temporary function called

```
Def add1(x):
    Return x+1
```

```
func = lambda x: x+1
```

```
Def make_power_func(n):
    Return lambda x:x**n
```

```
Square = make_power_func(2)
Print(square(3)) = 9
Cube = make_power_func(3)
Print(cube(2)) = 8
```

List - ordered, indexed, changeable, and allow duplicate values
Set - unordered, unchangeable, and unindexed, duplicates are not allowed
Tuple - ordered, indexed, unchangeable, and allow duplicate values
Dictionary - ordered, indexed by keys, changeable and do not allow duplicates. No duplicated, means cannot have two items with the same key (As a result, the values associated with those keys will override each other, and only the last assignment for each key will be retained.)
In a dictionary in Python, The values can be any data type, but the key cannot be list, dict, and set.

Ordered, means that the items have a defined order, and that order will not change.
Unchangeable, means that we cannot change, add or remove items after the tuple has been created.

List

List.append(x) → x is another element, *do not have to be a list, the element will be added at the back
List.remove() → removing the element that first appear in order, if element not in list, ERROR
List.sorted → will not modify the inputs
List.sort → will modify inputs

Tuples

An immutable list, you cannot directly call and change,
e.g., tuple[1] = 9 will give you error, neither can it append or remove
but you can still slice and index it, e.g., tup[1] or tup[1:4]
*Note: (3) is a int, (3,) is a tuple, no such thing as (,) or (,3)
Tuple.sorted works but not tuple.sort since tuple cannot be modified

Functions: Return returns a value, function without return would give a class of NoneType.

SetA = {1,2,3,4}
SetB = {3,4,5,6}

SetA | SetB	Union (combine both sets but only with element occurring once)	{1,2,3,4,5,6}
SetA & SetB	Intersection (common element)	{3,4}
SetA – SetB	A – B, Remove any element that is in A that is from B	{1,2}
SetA ^ SetB	(A|B)-A&B The order of function is Union and intersect first comes before minus	{1,2,5,6}

Dictionary

Dict1 = { key1 : value1, key2: value2}
To access the dictionary, dict1[key1] to find value1
Can change value in a key, dict1[key1] = value1000
YOU CANNOT ACCESS KEYS THAT DON'T EXISTS → ERROR
Dict1.get() → retrieve value and WONT CRASH if element don't exis
Dict1.clear() → clear all
Dict1.copy() → make a copy of dictionary
Dict1.keys() → return all keys
Dict1.values() → return all values
Dict1.item()→ return all keys + values
*Dictionary keys cannot be of type: list, set, dictionaries

Higher Order Functions:

e.g., 1 :
```
def f():
    print("hello")
def do_twice(x):
    x()
    x()

do_twice(f)
>> "hello"
>>"hello"
```

More on one-liners

```
print(list(str(123456)))
def digitsum(n):
    return sum(map(lambda x: int(x), list(str(n))))

print(digitsum(123345))

print("########")

def sds(n):
    return sum(map(lambda x: int(x) ** 2, list(str(n))))

print(sds(22222))
```

```
['1', '2', '3', '4', '5', '6']
18
########
20
```

```
def maxpower(n, exp):
    def count(m, cnt):
        if m >= exp:
            return cnt
        return count(m*n, 1+cnt)
    return count(1, 0)

print(maxpower(2, 1000))
print(maxpower(3, 1000))
```

```
10
7
```

The variables `n` and `exp` live in the scope of **count** but **not a local variable of count**

```
def partial(op):
    def action(a, b):
        print(op(a,b))
    return action

f1 = partial(lambda x, y : x * 2 + y * 2)

f1(5,10)
f2 = partial(f1)
f2(5,10)
```

```
30
30
None
```

The variable `op` live in the scope of **action** but **not local in action**?

map(function, sequence) → apply function to every element in sequence and RETURNS the element with function applied

```
tup = (1, -2, 3)
map1 = map(abs, tup)
map1 #map object
type(map1) #class map
map1list = list(map1)  # [1,2, 3]
map1tuple = tuple(map1)  #(), this is empty as it is already taken out of map1 for map1list. You can only convert to list/tuple once
```

DeepMapping:

```
def deepMap(func, seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return func(seq)
    else:
        return [deepMap(func, seq[0])] + deepMap(func, seq[1:])
```

filter(predicate, sequence) → Applies a function(predicate) that returns True or False to sequence's elements. Keep only the item if f(x) returns True, remove otherwise

```
l = [1, 2, 3, 'a', (1,2), ('b', 3)]
filter(lambda x:type(x) == int, l)  #filter object
list(filter(lambda x:type(x)==int, l) # [1, 2, 3]
```

f.read() → read whole file into a data as a string if there is new line it is inclusive of \n in the string

data.split() → split every data by the parameter, default is split everything by blank space

file type is also iterable
```
for a_line in f:
    print(a_line)
```

f.readline() → read the first line and then pointer goes to next line →will potray line as "abvcd\n"
If print(readline) → abvcd

f.readline().rstrip() → strip off any parameter at the back
f.readline().lstrip() → strip off any parameter at the front

Multidimensional Array / Matrix

```
def createZeroMatrix(r, c):
    output = []
    for i in range(r):
        row = []
        for j in range(c):
            row.append(0)
        output.append(row)
    return output

tic-tac-toe:
game = createZeroMatrix(3, 3)
for l in range(3):
    for j in range(3):
        game[i][j] = i*3+j+1
```

Nested Objects:
- Shallow Copy: Shares references to nested objects.
- Deep Copy: Creates independent copies of nested objects.

Use Cases:
- Shallow Copy: When you want a new object, but sharing nested objects is acceptable.
- Deep Copy: When you need a completely independent copy to avoid unintended changes.

Used as a one-liner

```
L = [9, 2, 1, 3, 4, 5, 6]

bigger_than_2 = map(lambda x: x > 2, L)
print(list(bigger_than_2))

remove_less_than_2 = list(filter(lambda x: x > 2, L))
print(remove_less_than_2)

odd_even = map(lambda x: 'o' if x % 2 else 'e', L)
print(list(odd_even))

odd_even_filter = list(filter(lambda x: 'o' if x % 2 else 'e', L))
print(odd_even_filter)

make_odd_string = map(str, list(filter(lambda x: x % 2, L)))
print(list(make_odd_string))

squared = str(list(filter(lambda x: x > 30, map(lambda x: x*x, L))))
print(squared)

squared_all = map(lambda x: x*x, L)
print(list(squared_all))
```

```
[True, False, False, True, True, True, True]
[9, 3, 4, 5, 6]
['o', 'e', 'o', 'o', 'e', 'o', 'e']
[9, 2, 1, 3, 4, 5, 6]
['9', '1', '3', '5']
[81, 36]
[81, 4, 1, 9, 16, 25, 36]
```

any()

Returns **True if at least one element of an iterable is true**. If the iterable is empty, it returns False.

```
any(iterable)
```

Parameter	Description
iterable	The iterable to be checked for truthiness

```
L = [1, 2, 3, 4]
result1 = any(x > 3 for x in L)
result2 = any(x > 9 for x in L)

def isPrime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

result3 = any(isPrime(x) for x in [4, 6, 8, 99])
result4 = any(isPrime(x) for x in [4, 6, 8, 97, 99])

print(result1)
print(result2)
print(result3)
print(result4)
```

```
True
False
False
True
```

Consecutive applications of function is treated as **left associative**.

```
(lambda x: x (lambda y: y))(lambda z : z)(1)
# (x (lambday: y) ({{ x → lambda z : z }}) (1)
# ((lambdaz : z) (lambday: y)) (1)
# (z ({{ z → lambda y : y }}) (1)
# (lambday : y) (1)
# y({{ y → 1 }})
# 1
```

```
1
```

map()

Executes a specified function **for each item in an iterable**. The item is sent to the function as a parameter.

```
map(function, iterables)
```

Parameter	Description
function	The function to execute for each item
iterable	A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

```
def myfunc(n):
    return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))
print(x)
print(list(x))
```

```
<map object at 0x000001115B6F0D0>
[5, 6, 6]
```

filter()

Returns an **iterator where the items are filtered through a function** to test if the item is accepted or not.

```
filter(function, iterable)
```

Parameter	Description
function	A function to be run for each item in the iterable
iterable	The iterable to be filtered

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
    if x < 18:
        return False
    else:
        return True

adults = filter(myFunc, ages)

print(adults)
print(list(adults))
```

```
<filter object at 0x000001115B6FB20>
[18, 24, 32]
```

OOPs (games):

```
Class StudentRecord():  #creating a parent class of student record
    def __init__(self):  #must def init in this way with self as para
        self.name = ''  #declare attributes that lasts
        name = ''  #note theres no self, hence disappear after
c                       completion of function __init__()

alan = StudentRecord()  #alan is now type class "StudentRecord"
alan.name = 'Alan'

class BankAccount():
    def __init__(self, name, balance): #making a parameter call
        self.name = name
        self.balance = balance

myAcc = BankAccount('John', 100)  #now need to call both parameters


Inheritance:

class Vehicle():                #Parent class
    def __init__(self, pos):
        self.pos = pos

class Car(Vehicle):             #chilren class with parent in para
    def __init__(self):
        super().__init__(pos) #it takes pos in Vehicle
```

Note:
1) Super() function takes parent function and copy the entirety of its method or attributes
2) Method defined in both parent and child class would carry out method in child first (e.g., both class have method of move(), but move() in child class takes precedence

Inheritance

Inheritance allows a class (subclass or derived class) to inherit the properties and methods of another class (base class or parent class). It promotes code reuse and the creation of a hierarchy of classes.

```
class HighSchoolStudent(Student):
    def __init__(self, name, age, grade, level):
        super().__init__(name, age, grade)
        self.level = level

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}, Level: {self.level}")

# Example of creating an instance of HighSchoolStudent
high_school_student = HighSchoolStudent("Shtface", 18, "B", 12)
high_school_student.display_info()
```

47. You are given a 2D map like the following. And you want to locate all the positions of the star by printing out the positions of their centers. Fill in the blanks in the following code to complete the function `find_star()`:

```
...
map1 = ['.#+#...#...',
        '..#...#+#...',
        '.......#....',
        '.+...#.....',
        '....#+#...#',
        '.....#...#+']
The stars are
 #
#+#
 #
>>> find_star(map1)
(1, 7)
(4, 5)
...
def find_star(m):
    n_row = len(m)
    n_col = len(m[0])
    for i in range(n_row):
        for j in range(1,n_col-1):
            if m[i][j] == '+' and m[i+1][j] == '#' and m[i-1][j] == '#' and m[i][j-1] == '#':
                print((i,j))

map1 = ['.#+#...#...',
        '..#...#+#...',
        '.......#....',
        '.+...#.....',
        '....#+#...#',
        '.....#...#+']

find_star(map1)
```

```
(1, 7)
(4, 5)
```

```
X = {} # Initialize an empty dictionary to store the computed results.
def nCk(n, k):
    if (n, k) in X:  # Check if the result is already computed
        return X[(n, k)]
    if n == k or k == 0:
        return 1
    ans = nCk(n-1, k-1) + nCk(n-1, k)
    X[(n, k)] = ans  # Memoize the result for future use
    return ans
```

all()

Returns **True if all elements of an iterable are true** (or if the iterable is empty). If any element is false, it returns False.

```
all(iterable)
```

Parameter	Description
iterable	The iterable to be checked for truthiness

```
L = [1, 2, 3, 4]
result5 = all(x > 3 for x in L)
result6 = all(x > 0 for x in L)

def isPrime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

result7 = all(isPrime(x) for x in [4, 6, 8, 9, 99])
result8 = all(isPrime(x) for x in [3, 5, 7, 11, 97])

print(result5)
print(result6)
print(result7)
print(result8)
```

```
False
True
False
True
```

Assert Statement

Definition:

- `assert` is used for debugging purposes to check whether a given condition is `True`, and if not, it raises an `AssertionError` exception.
- If the statement following in the assertion is `False` then `Exception` will be called.

Example:

```
x = 5
assert x > 0, "x should be a positive number"
print("Assertion passed!")
```

```
Assertion passed!
```

```
x = 5
assert x > 10, "x should be a positive number"
print("Assertion passed!")
```

```
x should be a positive number
```

Explanation:

- The `assert` statement checks if the given condition is `True`.
- If the condition is `False`, it raises an `AssertionError` with an optional error message.

```
while True:
    try:
        pos = int(input("Input: "))
        assert 0 < pos < 10
        break
    except AssertionError:
        print("Số không lớn hơn 10")
    except:
        print("Wrong")
```

```
Input: 1
```

reduce()

Applies a **rolling computation to sequential** pairs of values in an iterable

`reduce()` was a *built-in* function for python, but was later removed it [functools](#)

```
reduce(function, iterable[, initializer])
```

Parameter	Description
function	A function that takes two arguments and returns a value
iterable	The iterable to be reduced
initializer	(Optional) The initial value for the reduction

```
from functools import reduce

sum_all = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
combine_all = reduce(lambda x, y: x + y, ['a', 'b', 'c'])

print(sum_all)
print(combine_all)
```

```
15
abc
```

We can write our own reduce:

```
def reduce(function, iterable):
    if not iterable:
        return iterable
    first = iterable[0]
    for i in iterable[1:]:
        first = function(first, i)
    return first

sum_all = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
combine_all = reduce(lambda x, y: x + y, ['a', 'b', 'c'])

print(sum_all)
print(combine_all)
```

```
15
abc
```

isinstance() in Inheritance

In Python, the `isinstance()` function is utilized to check if an object belongs to a specified class or a tuple of classes. This method is particularly useful in scenarios involving inheritance, where you want to determine if an instance is derived from a specific class or its subclasses.

Consider the following scenario with a base class `Shape` and two subclasses `Circle` and `Square`:

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length ** 2
```

Now, let's create instances and use `isinstance()`:

```
circle_instance = Circle(5)
square_instance = Square(4)

print(isinstance(circle_instance, Shape))  # Output: True
print(isinstance(square_instance, Circle))  # Output: False
print(isinstance(square_instance, (Shape, Square)))  # Output: True
```

In this example, `isinstance()` is employed to check the type of objects, providing flexibility in handling instances of the base class and its subclasses.

This method is valuable for conditional statements and ensuring compatibility when dealing with objects derived from a common base class.

super() in Inheritance

In Python, `super()` is a built-in function that is used to refer to the superclass (parent class) of a derived class. It is commonly used in the `__init__` method of a subclass to invoke the constructor of the superclass.

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

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

class HighSchoolStudent(Student):
    def __init__(self, name, age, grade, level):
        super().__init__(name, age, grade)
        self.level = level

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}, Level: {self.level}")

high_school_student = HighSchoolStudent("Shtface", 18, "B", 12)
high_school_student.display_info()
```

```
Name: Shtface, Age: 18, Grade: B, Level: 12
```

Binary Search:
1) Find middle element
2) If is what we are looking for, return True
3) If our key is smaller than middle element, repeat search on LEFT (the smaller side) of the list
4) Else, repeat search on the right of list (search from middle again)

```
def binary_search(key, seq): # seq is sorted
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2 # get middle
        if key == seq[mid]:
            return True
        elif key < seq[mid]:
            return helper(low, mid-1)
        else:
            return helper(mid+1, high)
    return helper(0, len(seq)-1)
```

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)

L = [4, 5, 6, 7, 1, 2, 3, 9, 8]
print("Before sorting:", L)
sorted_list_quick = quick_sort(L)
print("After sorting:", sorted_list_quick)
```

Exceptions:
Errors coming from execution are exception errors:

1. ZeroDivisionError (e.g., 4/0)
2. NameError (name is not defined, such as 4 + name*3)
3. TypeError (e.g., 'cant convert 'int' object to str implicitly such as "2" +2)
4. Value Error (e.g., int('one')

To handle: add exceptions

1. Include `Try` Clause
2. If an exception occurred, skip the rest of the try clause, to a matching except clause
3. If no exception occurs, the except clause is skipped (go to the else clause, if it exists)
4. The `finally` clause is always executed before leaving the try statement, whether an exception has occurred or not.

Example:

```
try:
    # code that may raise an exception
    result = 10 / 0
except ZeroDivisionError as e:
    # code to handle the exception
    print(f"Error: {e}")
else:
    # code to be executed if no exception occurs
    print("No error occurred.")
finally:
    # code to be executed regardless of whether an exception occurred or not
    print("This will always be executed.")
```

```
Error: division by zero
This will always be executed.
```

Explanation:

- The `try` block contains the code that may raise an exception.
- The `except` block catches and handles the specified exception.
- The `else` block is executed if no exception occurs in the `try` block.
- The `finally` block contains code that will always be executed, whether an exception occurred or not.

Inside the scope

In Python, the `try` block will catch an exception if it occurs anywhere within its scope, including inside the `while` loop. The `while` loop continues to execute until the condition `n > 0` is true. Once `n` becomes 0, the loop exits, and the program proceeds to the `except` block.

```
try:
    n = 10
    while n > 0:
        n -= 1
        if n == 5:
            raise ValueError("n is 5, raising an exception")
except ValueError as e:
    print("Caught an exception:", e)
```

```
Caught an exception: n is 5, raising an exception
```

(catches any exceptions that may occur during its execution, and the loop runs until `n` reaches 0. the `while` loop exits, and the program goes to the `except` block.)

Memoization

```
fibans = {}
def fibm(n):
    if n in fibans.keys():
        return fibans[n]
    if (n == 0):
        ans = 0
    elif (n == 1):
        ans = 1
    else:
        ans = fibm(n - 1) + fibm(n - 2)
    fibans[n] = ans
    return ans
```

- Create a dictionary to remember the answer if `fibm(n)` is computed before
- If the ans was computed before, get the answer direction from the dictionary
- Otherwise, compute the ans and put it into the dictionary for later use

Recursion Removal

```
def fibi(n):
    ans = [0,1,1]
    if n < 3:
        return ans[n]
    for i in range(3,n+1):
        ans.append(ans[i-1]+ans[i-2])
    return ans[n]
```

- Store all the answers in an array
- Add a new fib(i)
  - as fib(i-1) + fib(i-2)
- Wait a min...
  - Do we need all the past numbers if we only need fib(n)?

Recursion Removal 2

- Add a new fib(i)
  - as fib(i-1) + fib(i-2)
- And I only need to keep fib(i-1) and fib(i-2)

```
def fibi2(n):
    if n < 3:
        return 1
    fibminus1,fibminus2 = 1,1
    for i in range(3,n+1):
        fibminus2,fibminus1 = fibminus1, fibminus1 + fibminus2
    return fibminus1
```

CS1010E Exams Cheat Sheet
Brians Tjipto
found: https://github.com/brianstm/NUS