In [ ]:
```
# Topics:
- For Loop and Break
- None in Python
- Assignment, Object and References in Python
- Number Basics
- Function Scope
- Exception Handling
- Lambda
- Deque
- NameTuple
```

For loop and break

In [2]:
```
input = [1,2,3,4,7,6]

res = ''

for num in input:
    if num == 5:
        res = 'found'
        break
else:
    res = 'no found'

print(res)
```

```
no fond
```

In [3]:
```
input = [[1,2,3,4,7,6],[4,7],[5,8]]

res = []

for group in input:
    for num in group:
        if num == 5:
            res.append('found')
            break
    else:
        res.append('no found')

print(res)
```

```
['no found', 'no found', 'found']
```

Null in Python: None

Python uses the keyword **None** to define *null* objects and variables. As the *null* in Python, **None** is not defined to be 0 or any other value. In Python, **None** is an *object* and a first-class citizen

None is the value a function returns when there is no return statement in the function:

```
>>> def has_no_return():
...     pass
>>> has_no_return()
```

```
>>> print(has_no_return())
None
```

An important rule to keep in mind when you're checking for None:

- Do use the identity operators is and is not.
- Do not use the equality operators == and !=.

The following objects are all falsy as well:

- None
- Empty lists
- Empty dictionaries
- Empty sets
- Empty strings
- 0
- False

Q: Does 'None' bind to a specific memory location?

Q: `m=None  n=None` , does `m=n` ?

Q:

```
dummy.next = m = None
...
m = ListNode(4)
```

So does `dummy.next = ListNode(4)` ?

In [15]:
```
m = None
print(id(m))
n = None
print(id(n))

"""
None is a singleton. That is, the NoneType class only ever gives you the same si
There's only one None in your Python program
"""
```

```
4390759864
4390759864
```

Out[15]:
```
'\nNone is a singleton. That is, the NoneType class only ever gives you the same
single instance of None. \nThere's only one None in your Python program\n'
```

In many other languages, null is just a synonym for 0, but null in Python is a full-blown object:

```
>>> type(None)
<class 'NoneType'>
```

In some languages, variables come to life from a **declaration**. They don't have to have an initial value assigned to them. In those languages, the initial default value for some types of variables might be null. In Python, however, variables come to life from **assignment statements**.

```
>>> print(bar)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
>>> bar = None
>>> print(bar)
None
```

Assignment, Object and References in Python

In [13]:
```
"""
The built-in Python function id() returns an object's integer identifier.
Using the id() function, you can verify that two variables indeed point to the s
"""

m = 300
n = 300
print(id(m))
print(id(n))
```

```
4430456048
4430455984
```

In [9]:
```
"""
With the statement m = 300, Python creates an integer object with the value 300
n is then similarly assigned to an integer object with value 300—but not the sam
"""

>>> m = 300
>>> n = 300
>>> id(m)
60062304
>>> id(n)
60062896
```

Out[9]:
```
60062896
```

In [ ]:
```
# But

>>> m = 30
>>> n = 30
>>> id(m)
1405569120
>>> id(n)
1405569120

"""
For purposes of optimization, the interpreter creates objects for the integers i
at startup, and then reuses them during program execution. Thus, when you assign
integer value in this range, they will actually reference the same object.
"""
```

Number Basics

What is the maximum possible value of an integer in

Python ?

In Python, value of an integer is not restricted by the number of bits and can expand to the limit of the available memory (Sources : this and this). Thus we never need any special arrangement for storing large numbers (Imagine doing above arithmetic in C/C++).

As a side note, in Python 3, there is only one type “int” for all type of integers. In Python 2.7. there are two separate types “int” (which is 32 bit) and “long int” that is same as “int” of Python 3.x, i.e., can store arbitrarily large numbers.

```
# A Python program to show that there are two types in
# Python 2.7 : int and long int
# And in Python 3 there is only one type : int

x = 10
print(type(x))

x = 10000000000000000000000000000000000000000000000
print(type(x))

# Python 2
<type 'int'>
<type 'long'>

# Python 3
<type 'int'>
<type 'int'>
```

In [1]:
```
x = 10
print(type(x))

x = 10000000000000000000000000000000000000000000000
print(type(x))
```

```
<class 'int'>
<class 'int'>
```

In [3]:
```
import sys
print(sys.maxsize)
```

```
9223372036854775807
```

In [16]:
```
help(divmod)
```

```
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

Round int

- general round()

- always round down
- round up

```
>>> import math
>>> math.floor(12.6)  # returns 12.0 in Python 2
12
>>> int(12.6)
12
>>> math.trunc(12.6)
12
```

However, note that they behave differently with negative numbers: int and math.trunc will go to 0, whereas math.floor always floors downwards

```
>>> import math
>>> math.floor(-12.6)  # returns -13.0 in Python 2
-13
>>> int(-12.6)
-12
>>> math.trunc(-12.6)
-12
```

Tricky points about round()

```
>>> round(2.5)
2

>>> round(1.5)
2
```

Another example

```
print(round(2.665, 2))
print(round(2.675, 2))
>>>2.67
>>>2.67
```

Check if a number is integer or decimal in Python

```
print(isinstance(i, int))
# True

print(isinstance(i, float))
# False

print(isinstance(f, int))
# False

print(isinstance(f, float))
# True
```

Check if *float* is integer

```
f = 1.23
```

```
print(f.is_integer())
# False

f_i = 100.0

print(f_i.is_integer())
# True
```

Check if string is integer

```
def is_integer(n):
    try:
        float(n)
    except ValueError:
        return False
    else:
        return float(n).is_integer()

print(is_integer(100))
# True

print(is_integer(100.0))
# True

print(is_integer(1.23))
# False

print(is_integer('100'))
# True

print(is_integer('100.0'))
# True

print(is_integer('1.23'))
# False

print(is_integer('string'))
# False
```

Float and scientic notation

In [8]:

```
ff = float(0.0519**-4)
print(ff)
g = 0.00001357
print(g)
print('using format str g:{:f}'.format(g))
```

```
137825.81479394066
1.357e-05
using format str g:0.000014
```

Ascii in Python

Unicode is a superset of ASCII, and the numbers 0–128 have the same meaning in ASCII as they have in Unicode.

ord(c)

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, ord('a') returns the integer 97 and ord('€') (Euro sign) returns 8364. This is the inverse of chr().

```
ord('A')  #--> 65
ord('B')  #--> 66
ord('C')  #--> 67
ord('a')  #--> 97
```

Reverse of ord() is chr()

```
chr(65)  #--> 'A'
chr(127) #--> '\x7f' memory address of [DEL] in hex
```

If...Else...

Simplier way to write:

```
if flag > 0:
    num = ...
else:
    num = ...


num = ... if flag>0 else ...
```

Function Basics

In [13]:
```
def outter():
    a = 12
    # count() # used after define count()
    def count():
        for i in range(a):
            print(i)
    count()
outter()
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

In [19]:
```
def outter():
    b = 12

    def count():
        global b  # b should be declared outside of outter()
        for i in range(b):
            print(i)
    count()
outter()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-c7140fc7a9e5> in <module>
      9             print(i)
     10     count()
---> 11 outter()

<ipython-input-19-c7140fc7a9e5> in outter()
      8         for i in range(b):
      9             print(i)
---> 10     count()
     11 outter()

<ipython-input-19-c7140fc7a9e5> in count()
      6     def count():
      7         global b
----> 8         for i in range(b):
      9             print(i)
     10     count()

NameError: name 'b' is not defined
```

In [20]:
```
b=12
def outter():
#     b = 12

    def count():
        global b  # b should be declared outside of outter()
        for i in range(b):
            print(i)
    count()
outter()
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

In [21]:
```
def outter():
    c=12

    def count():
```

```
        # global c

        for i in range(c):  # actually you don't need to declare global c
            print(i)
    count()
outter()
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

In [32]:

```
def outter():
    d=5

    def count():
        nonlocal d
        d=8
        for i in range(d):  # actually you don't need to declare global c
            print(i)
    count()
    print("d:", d) # nonlocal makes d come up from inner scope, covering d=5
outter()
```

```
0
1
2
3
4
5
6
7
d: 8
```

Exception Handling

Tutorial 1

In [1]:

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occured.")
        print("Next entry.")
```

```
        print()
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```

In [ ]:

```
# Using raise

class Solution:
    def isCousins(self, root: TreeNode, x: int, y: int) -> bool:
        xh = yh = -10

        def getHeight(node):
            nonlocal xh, yh
            if not node:
                return 0

            lh = getHeight(node.left)
            rh = getHeight(node.right)
            height = 0

            if lh > rh:
                height = lh+1
            else:
                height = rh+1

            if xh > 0 and yh > 0:
                if node.left and node.right and \
                    ((node.left.val == x and node.right.val == y) or \
                    (node.right.val == y and node.right.val == x)):
                    # not cousin
                    raise Exception("No")
                if xh == yh:
                    # is cousin
                    raise Exception("Yes")
                else:
                    # not counsin
                    raise Exception("No")

            if node.val == x:
                xh = height
            if node.val == y:
                yh = height

            return height

        try:
            ret = getHeight(root)
        except Exception as exc:
            if str(exc) == 'Yes':
                return True
            else:
                return False
```

```
        return False
```

Lambda

ref Definition

```
>>> lambda x: x
```

- The keyword: lambda
- A bound variable: x
- A body: x

Use `lambda` like an expression

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

>>> (lambda x, y: x + y)(2, 3)
5

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

Check the inside of `lambda` and regular function

```
>>> import dis
>>> add = lambda x, y: x + y
>>> type(add)
<class 'function'>
>>> dis.dis(add)
  1           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (y)
              4 BINARY_ADD
              6 RETURN_VALUE
>>> add
<function <lambda> at 0x7f30c6ce9ea0>
```

Traceback

Arguments

```
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
>>> (lambda *args: sum(args))(1,2,3)
6
>>> (lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3)
6
```

```
>>> (lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3)
6
```

Closure

```
def outer_func(x):
    y = 4
    return lambda z: x + y + z

for i in range(3):
    closure = outer_func(i)
    print(f"closure({i+5}) = {closure(i+5)}")
```

Appropriate Uses of Lambda Expressions

- combined with `map()` , `filter()` , `reduce`

```
>>> list(map(lambda x: x.upper(), ['cat', 'dog', 'cow']))
['CAT', 'DOG', 'COW']
>>> list(filter(lambda x: 'o' in x, ['cat', 'dog', 'cow']))
['dog', 'cow']
>>> from functools import reduce
>>> reduce(lambda acc, x: f'{acc} | {x}', ['cat', 'dog', 'cow'])
'cat | dog | cow'
```

- key function in `.sort()` and `sorted()`

```
>>> ids = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']
>>> print(sorted(ids)) # Lexicographic sort
['id1', 'id100', 'id2', 'id22', 'id3', 'id30']
>>> sorted_ids = sorted(ids, key=lambda x: int(x[2:])) # Integer sort
>>> print(sorted_ids)
['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

- UI framework
- Strongly suggest not to write lambda function with > 1 line
  https://stackoverflow.com/questions/1233448/no-multiline-lambda-in-python-why-not

Deque

In [ ]:

```
Python implements deque by doubly linked list. So
- Deque is easier to extend than list
- Deque[3] is slow as the linked list needs to be traversed
```

pop() Remove and return an element from the right side of the deque. If no elements are present, raises an **IndexError**.

popleft() Remove and return an element from the left side of the deque. If no elements are present, raises an **IndexError**.

In [1]:

```
# https://www.geeksforgeeks.org/deque-in-python/
# https://docs.python.org/3.9/library/collections.html#collections.deque
```

```
from collections import deque

history = deque()

history.append('China')
history.append('Britain')
history.append('Greece')
history
```

Out[1]: deque(['China', 'Britain', 'Greece'])

In [2]:
```
print(history[0])
print(len(history))
```

```
China
3
```

In [5]:
```
nations = deque()

nations.appendleft('China')
nations.appendleft('Korea')
nations.appendleft('USA')
nations
```

Out[5]: deque(['USA', 'Korea', 'China'])

In [6]:
```
nations[0]
```

Out[6]: 'USA'

NameTuples

- A dictionary but can also be accessed by indexes

```
# Python code to demonstrate namedtuple()

from collections import namedtuple

# Declaring namedtuple()
Student = namedtuple('Student',['name','age','DOB'])

# Adding values
S = Student('Nandini','19','2541997')

# Access using index
print ("The Student age using index is : ",end ="")
print (S[1])

# Access using name
print ("The Student name using keyname is : ",end ="")
print (S.name)
```

```
# Access using getattr()
print ("The Student DOB using getattr() is : ",end ="")
print (getattr(S,'DOB'))


# ----------------------
# Output
The Student age using index is : 19
The Student name using keyname is : Nandini
The Student DOB using getattr() is : 2541997
```

- _make() :- This function is used to return a namedtuple() from the iterable passed as argument.
- _asdict() :- This function returns the OrderedDict() as constructed from the mapped values of namedtuple().
- using “**” (double star) operator :- This function is used to convert a dictionary into the namedtuple().

```
# Python code to demonstrate namedtuple() and
# _make(), _asdict() and "**" operator

# importing "collections" for namedtuple()
import collections

# Declaring namedtuple()
Student = collections.namedtuple('Student',['name','age','DOB'])

# Adding values
S = Student('Nandini','19','2541997')

# initializing iterable
li = ['Manjeet', '19', '411997' ]

# initializing dict
di = { 'name' : "Nikhil", 'age' : 19 , 'DOB' : '1391997' }

# using _make() to return namedtuple()
print ("The namedtuple instance using iterable is  : ")
print (Student._make(li))

# using _asdict() to return an OrderedDict()
print ("The OrderedDict instance using namedtuple is  : ")
print (S._asdict())

# using ** operator to return namedtuple from dictionary
print ("The namedtuple instance from dict is  : ")
print (Student(**di))


# ----------------------
# Output
The namedtuple instance using iterable is  :
```

```
Student(name='Manjeet', age='19', DOB='411997')
The OrderedDict instance using namedtuple is  :
OrderedDict([('name', 'Nandini'), ('age', '19'), ('DOB', '2541997')])
The namedtuple instance from dict is  :
Student(name='Nikhil', age=19, DOB='1391997')
```

In [ ]: