009_Python_Data_Types

March 2, 2023

1 Python Data Types

In this class, you will learn about different data types you can use in Python.

1.1 Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

1.2 1. Python Numbers

Integers, floating point numbers and complex numbers fall under **Python numbers** category. They are defined as **int**, **float** and **complex** classes in Python.

We can use the **type()** function to know which class a variable or a value belongs to.

Similarly, the **isinstance()** function is used to check if an object belongs to a particular class.

[1]:

```
a = 6
print(a, "is of type", type(a))
print(a, "is integer number?", isinstance(5,int))

a = 3.0
print(a, "is of type", type(a))
print(a, "is float number?", isinstance(2.0,float))

a = 1+2j  # '1' is real part and '2j' is imaginary part
print(a, "is of type", type(a))
print(a, "is complex number?", isinstance(1+2j,complex))
```

```
6 is of type <class 'int'>
6 is integer number? True
3.0 is of type <class 'float'>
3.0 is float number? True
(1+2j) is of type <class 'complex'>
(1+2j) is complex number? True
```

Integers can be of any length, it is only limited by the memory available.

A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. `1` is an integer, `1.0` is a floating-point number.

Complex numbers are written in the form, `x + yj`, where `x` is the real part and `y` is the imaginary part. Here are some examples.

[2]:
```
a = 1234567890123456789
print (a)

b = 0.1234567890123456789  # total of only 17 numbers after decimal can be␣
 ↪printed.
print (b)

c = 1+2j
print (c)
```

```
1234567890123456789
0.12345678901234568
(1+2j)
```

Notice that the `float` variable `b` got truncated.

1.3 2. Python List []

List is an **ordered sequence** of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets `[ ]`.

```
>>>a = [1, 3.3, 'python']
```

We can use the slicing operator `[ ]` to extract an item or a range of items from a list. The index starts from 0 in Python.

[3]:
```
x = [6, 99, 77, 'Apple']
print(x, "is of type", type(x))
```

```
[6, 99, 77, 'Apple'] is of type <class 'list'>
```

[4]:
```
a = [5, 10, 15, 20, 25, 30, 35, 40]  # Total elemnets is 8
#   [0   1   2   3   4   5   6   7]    Index forward
#   [-8 -7  -6  -5  -4  -3  -2  -1]    Index backward

# index '0' is element '1' = 5,
# index '1' is element '2' = 10,
# index '2' is element '3' = 15,
# .
# .
# .
# index '7' is element '8' = 40,
a[1] # To access the elements in the list

# a[2] = 15
print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]
print("a[0:3] = ", a[0:3])  # [0:3] means elements from 0 uptil 2 index (not␣
 ↪include last element)
                            # [0:3] means from index 0 to 3 - 1
                            # [0:3] means from index 0 to 2

# a[5:] = [30, 35, 40]  # [5:] means all the elements from 5 till end
print("a[5:] = ", a[5:])
```

```
a[2] =  15
a[0:3] =  [5, 10, 15]
a[5:] =  [30, 35, 40]
```

[5]:
```
a[1:-2]
```

[5]:
```
[10, 15, 20, 25, 30]
```

[6]:
```
a[5:9]
```

[6]:
```
[30, 35, 40]
```

[7]:
```
a[:5]
```

[7]:
```
[5, 10, 15, 20, 25]
```

Lists are **mutable**, meaning, the value of elements of a list can be altered.

[8]:
```
# Change the element of the List
a = [1, 2, 3]
#    [0  1  2]   Index forward

a[2] = 4  # Change my third element from '3' to '4' # [2] is the index number
print(a)
```

```
[1, 2, 4]
```

1.4 3. Python Tuple ()

Tuple is an **ordered sequence** of items same as a list. The only difference is that tuples are **immutable**. Tuples once created cannot be modified.

Tuples are used to **write-protect data** and are usually faster than lists as they cannot change dynamically.

It is defined within parentheses `()` where items are separated by commas.

```
>>>t = (6,'program', 1+3j)
```

We can use the slicing operator `[]` to extract items but we cannot change its value.

[9]:
```
# Tuple 't' have 3 elements
t = (6,'program', 1+3j)
#   (0        1      2)   Index forward

# index '0' is element '1'= 6
# index '1' is element '2'= program
# index '2' is elemtnt '3'= 1+3j

# t[1] = 'program'
print("t[1] = ", t[1])

# t[0:3] = (6, 'program', (1+3j))
print("t[0:3] = ", t[0:3])

# Generates error
# Tuples are immutable
t[0] = 10  # trying to change element 0 from '6' to '10'
```

```
t[1] =  program
t[0:3] =  (6, 'program', (1+3j))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-c92d9f2b36de> in <module>
     15 # Generates error
     16 # Tuples are immutable
---> 17 t[0] = 10  # trying to change element 0 from '6' to '10'

TypeError: 'tuple' object does not support item assignment
```

[10]:
```
list1 =  [9, 'apple', 3 + 6j]  # list
tuple1 = (9, 'apple', 3 + 6j)  # tuple

list1[1] = 'banana'  # List is mutable
print(list1)         # No error
tuple1[1]= 'banana'  # Tuple is immutable
print(tuple1)        # error
```

```
[9, 'banana', (3+6j)]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-10-e32e417070a1> in <module>
      4 list1[1] = 'banana'  # List is mutable
      5 print(list1)         # No error
----> 6 tuple1[1]= 'banana'  # Tuple is immutable
      7 print(tuple1)        # error

TypeError: 'tuple' object does not support item assignment
```

1.5 4. Python Strings

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, `'''` or `"""`.

[11]:

```
s = '''Apple'''
print(s)
s = """Apple"""
print(s)
s = 'Apple'
print(s)
s = "Apple"
print(s)
s = Apple   # cannot write string with out quotes ('', " ", """ """, ''' ''')
print(s)
```

```
Apple
Apple
Apple
Apple
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-7fce570bf337> in <module>
      7 s = "Apple"
      8 print(s)
----> 9 s = Apple   # cannot write string with out quotes ('', " ", """ """, ''
 ↪''')
     10 print(s)

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

[12]:

```
s = "This is a string"  # s is my variable
print(s)
s = '''A multiline
string'''
print(s)
```

```
This is a string
```

```
A multiline
string
```

Just like a list and tuple, the slicing operator `[ ]` can be used with strings. Strings, however, are **immutable**.

[13]:
```
s = 'Hello world!' # total 12 elements. Index start from '0' to '11'

# s[4] = 'o'
print("s[4] = ", s[4])

# s[6:11] = 'world' # index '6' to '11' means element from 6 to 10
print("s[6:11] = ", s[6:11])
```

```
s[4] =  o
s[6:11] =  world
```

[14]:
```
a = "apple"

a[0]='o'

# Simiar to TUPLE, STRING is immutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-466b90e7ef2f> in <module>
      1 a = "apple"
      2
----> 3 a[0]='o'
      4
      5 # Simiar to TUPLE, STRING is immutable

TypeError: 'str' object does not support item assignment
```

1.6 5. Python Set {}

Set is an **unordered collection** of unique items. Set is defined by values separated by comma inside braces `{ }`. Items in a set are not ordered.

[15]:
```
a = {7,1,3,6,9}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))
```

```
a =  {1, 3, 6, 7, 9}
<class 'set'>
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

```
[16]: a = {1,2,2,3,3,3} # we can see total 6 elements
      print(a)
```

```
{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator [] does not work.

```
[17]: a = {1,2,3}  # in Set data type we cannot access the elements because set is␣
       ↪unordered collection
      a[1]  # Index [1] means element 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-242b77ef2a87> in <module>
      1 a = {1,2,3}  # in Set data type we cannot access the elements because␣
 ↪set is unordered collection
----> 2 a[1]  # Index [1] means element 2

TypeError: 'set' object is not subscriptable
```

1.7 6. Python Dictionary {}

Dictionary is an **unordered collection** of **key-value pairs**.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
[18]: d = {1: 'Apple', 2: 'Cat', 3: 'Food'}  # 'Apple' is element and 1 is the key of␣
       ↪element.
      print(d, type(d))

      d[3]
```

```
{1: 'Apple', 2: 'Cat', 3: 'Food'} <class 'dict'>
```

```
[18]: 'Food'
```

```
[19]: d = {1:'value','key':2} # d is my variable, 'value' and 'key' are the element␣
       ↪and 1 and 2 are the key.
```

```
type(d)
```

```
[19]: dict
```

We use key to retrieve the respective value. But not the other way around.

```
[20]: d = {1:'value','key':2} # '1' is the key to access 'value' and 'key' is the key␣
        ↪to access '2'
      print(type(d))

      print("d[1] = ", d[1]); # try to find the element from key.

      print("d['key'] = ", d['key']);  # try to find the key from the element.
```

```
<class 'dict'>
d[1] =  value
d['key'] =  2
```

```
[21]: print(type(zip([1,2],[3,4])))
```

```
<class 'zip'>
```

```
[ ]:
```

012_Python_Operators

March 2, 2023

1 Python Operators

Python can be used like a calculator. Simply type in expressions to get them evaluated.

What are operators in python?

Operators are special **symbols** in Python that carry out **arithmetic** or **logical computation**. The value that the operator operates on is called the **operand**.

For example:

```
>>>6+3
```

```
9
```

Here, `+` is the operator that performs addition. `2` and `3` are the operands and `5` is the output of the **operation**.

```
[1]: 6+3
```

```
[1]: 9
```

1.1 1. Arithmatic Operators

Arithmetic operators are used to perform **mathematical operations** like **addition**, **subtraction**, **multiplication** etc.

Symbol	Task Performed	Meaning	Example
`+`	Addition	add two operands or unary plus	**x + y** or **+2**
`-`	Subtraction	substract right operand from the left or unary minus	**x - y** or **-2**
`*`	Multiplication	Multiply two operands	**x * y**

Symbol	Task Performed	Meaning	Example
/	Division	Divide left operand by the right one (always results into float)	**x** / **y**
%	Modulus (remainder)	remainder of the division of left operand by the right	**x** % **y** (remainder of **x**/**y**)
//	Integer/Floor division	division that results into whole number adjusted to the left in the number line	**x** // **y**
**	Exponentiation (power)	left operand raised to the power of right	**x** ** **y** (**x** to the power **y**)

As expected these operations generally promote to the most general type of any of the numbers involved i.e. int -> float -> complex.

Example : Arithmetic operators in Python

[2]:

```
print('Addition: ', 1 + 2)
print('Subtraction: ', 2 - 1)
print('Multiplication: ', 2 * 3)
print ('Division: ', 4 / 2)                        # Division in python gives␣
 ↪floating number
print('Division: ', 6 / 2)
print('Division: ', 7 / 2)
print('Division without the remainder: ', 7 // 2)   # gives without the␣
 ↪floating number or without the remaining
print('Modulus: ', 3 % 2)                           # Gives the remainder
print ('Division without the remainder: ',7 // 3)
print('Exponential: ', 3 ** 2)                     # it means 3 * 3
```

```
Addition:  3
Subtraction:  1
Multiplication:  6
Division:  2.0
Division:  3.0
Division:  3.5
Division without the remainder:  3
Modulus:  1
Division without the remainder:  2
Exponential:  9
```

[3]:
```
x = 16
y = 3

print('x + y =',x+y) # 19
print('x - y =',x-y) # 13
print('x * y =',x*y) # 48
print('x / y =',x/y) # 5.333
print('x // y =',x//y) # 519
```

```
x + y = 19
x - y = 13
x * y = 48
x / y = 5.333333333333333
x // y = 5
```

[4]:
```
1+2+3
```

[4]:
```
6
```

[5]:
```
7-1
```

[5]:
```
6
```

[6]:
```
6 * (3+0j) * 1.0
```

[6]:
```
(18+0j)
```

[7]:
```
5/6
```

[7]:
```
0.8333333333333334
```

In many languages (and older versions of python) $\frac{1}{2} = 0$ (truncated division). In Python 3 this behaviour is captured by a separate operator that rounds down: (i.e., **a** // **b**= $\lfloor\frac{a}{b}\rfloor$)

[8]:
```
5//6.0
```

[8]:
```
0.0
```

[9]:
```
15%10
```

[9]:
```
5
```

[10]:
```
3 ** 2      # it means 3 * 3
```

[10]:
```
9
```

Python natively allows (nearly) infinite length integers while floating point numbers are double precision numbers:

```
[11]: 22**600
```

```
[11]: 28418980453852622603883856494709731311218158362293224700445543940796643776911788
      12950857724630992925806959846789600828759605386308210974680194772228078372003264
      98181984644523753583486191200211357972518570195570325656386596677454211151944685
      59021541269027438746788486392649581405327516658718317011453858465218779880972140
      61255150414970108216417936748419857073423000352835582654564422379477991172391711
      58452137399213278204090054607085479424452349064988805884112767551200021089635109
      55276534630522965966177951626038063994937181228938844252668098515385045718428663
      54778812217379870471751097931974983408341900435522201204371585512361524968203958
      78444312462690176127282917107188213888406272774071898635097899293391024066551157
      86899696829902032282666983426897111750208839152830573688512831984602085360572485
      861376
```

```
[12]: 22.0**600
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-12-bfc5aa62a0ff> in <module>
----> 1 22.0**600

OverflowError: (34, 'Result too large')
```

1.2 2. Comparison/Relational operators

Comparison operators are used to **compare values**. It either returns **True** or **False** according to the **condition**.

Symbol	Task Performed	Meaning	Example
`>`	greater than	True if left operand is greater than the right	**x > y**
`<`	less than	True if left operand is less than the right	**x < y**
`==`	equal to	True if both operands are equal	**x == y**
`!=`	not equal to	True if both operands are not equal	**x != y**
`>=`	greater than or equal to	True if left operand is greater than or equal to the right	**x >= y**

Symbol	Task Performed	Meaning	Example
<=	less than or equal to	True if left operand is less than or equal to the right	**x <= y**

Note the difference between **==** (equality test) and **=** (assignment)

Example : Comparison operators in Python

[13]:

```python
print(6 > 3)                            # True, because 3 is greater than 2
print(6 >= 3)                           # True, because 3 is greater than 2
print(6 < 3)                            # False,  because 3 is greater than 2
print(3 < 6)                            # True, because 2 is less than 3
print(3 <= 6)                           # True, because 2 is less than 3
print(6 == 3)                           # False, because 3 is not equal to 2
print(6 != 3)                           # True, because 3 is not equal to 2
print(len("apple") == len("avocado"))  # False
print(len("apple") != len("avocado"))  # True
print(len("apple") < len("avocado"))   # True
print(len("banana") != len("orange"))  # False
print(len("banana") == len("orange"))  # True
print(len("tomato") == len("potato"))  # True
print(len("python") > len("coding"))   # False
```

```
True
True
False
True
True
False
True
False
True
True
False
True
True
False
```

[14]:

```python
x = 30
y = 22

print('x > y is',x>y)   # False
print('x < y is',x<y)   # True
print('x >= y is',x>=y) # False
print('x <= y is',x<=y) # True
```

```
x > y is True
x < y is False
x >= y is True
x <= y is False
```

[15]:

```
z = 3  # 3 is assign to variable z
z == 3 # 3 is equal to z
```

[15]:

```
True
```

[16]:

```
z > 3
```

[16]:

```
False
```

Comparisons can also be chained in the mathematically obvious way. The following will work as expected in Python (but not in other languages like C/C++):

[17]:

```
0.5 < z <= 1   # z == 3
```

[17]:

```
False
```

1.3 3. Logical/Boolean operators

Logical operators are the **and**, **or**, **not** operators.

Symbol	Meaning	Example
and	True if both the operands are true	**x and y**
or	True if either of the operand is true	**x or y**
not	True if operand are false (complements the operand)	**not x**

Example : Logical operators in Python

[18]:

```
print('True == True: ', True == True)
print('True == False: ', True == False)
print('False == False:', False == False)
print('True and True: ', True and True)
print('True or False:', True or False)
```

```
True == True:  True
True == False:  False
False == False: True
True and True:  True
True or False: True
```

[19]:

```
# Another way comparison

print('1 is 1', 1 is 1)                    # True  - because the data values are␣
 ↪the same
print('1 is not 2', 1 is not 2)            # True  - because 1 is not 2
print('A in Milaan', 'A' in 'Milaan')      # True  - A found in the string
print('B in Milaan', 'B' in 'Milaan')      # False - there is no uppercase B
print('python' in 'python is fun')         # True  - because coding for all has␣
 ↪the word coding
print('a in an:', 'a' in 'an')             # True
print('27 is 3 ** 3:', 27 is 3**3)         # True
```

```
1 is 1 True
1 is not 2 True
A in Milaan False
B in Milaan False
True
a in an: True
27 is 3 ** 3: True
```

```
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:9: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is not" with a literal. Did you mean "!="?
<>:9: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-19-7c9145eb11e9>:3: SyntaxWarning: "is" with a literal. Did you
mean "=="?
  print('1 is 1', 1 is 1)                    # True  - because the data values are
the same
<ipython-input-19-7c9145eb11e9>:4: SyntaxWarning: "is not" with a literal. Did
you mean "!="?
  print('1 is not 2', 1 is not 2)            # True  - because 1 is not 2
<ipython-input-19-7c9145eb11e9>:9: SyntaxWarning: "is" with a literal. Did you
mean "=="?
  print('27 is 3 ** 3:', 27 is 3**3)         # True
```

[20]:

```
print(6 > 3 and 5 > 3) # True  - because both statements are true
print(6 > 3 and 5 < 3) # False - because the second statement is false
print(6 < 3 and 5 < 3) # False - because both statements are false
print(6 > 3 or 5 > 3)  # True  - because both statements are true
print(6 > 3 or 5 < 3)  # True  - because one of the statement is true
print(6 < 3 or 5 < 3)  # False - because both statements are false
print(not 6 > 3)       # False - because 6 > 3 is true, then not True gives␣
 ↪False
print(not True)        # False - Negation, the not operator turns true to false
print(not False)       # True
print(not not True)     # True
print(not not False)    # False
```

```
True
False
False
True
True
False
False
False
True
True
False
```

[21]:

```
x = True
y = False

print('x and y is',x and y) # False
print('x or y is',x or y) # True
print('not x is',not x) # False
```

```
x and y is False
x or y is True
not x is False
```

[22]:

```
True and (not(not False)) or (True and (not True))  # What will be output?

# True and (not(True)) or (True and (False))
# True and False or (False)
# False or False
# False
```

[22]: `False`

Here is the **truth table (@ and, or, not)** for these operators.

1.4 4. Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates **bit by bit**, hence the name.

For example: 2 is `10` in binary and 7 is `111`.

In the table below: Let **x** = 10 (`0000 1010` in binary) and **y** = 4 (`0000 0100` in binary)

Operator	Meaning	Symbol	Task Performed	Example
and	Logical and	**&**	Bitwise And	**x** & **y** = 0 (0000 0000)
or	Logical or	|	Bitwise OR	**x** | **y** = 14 (0000 1110)
not	Not	**~**	Bitwise NOT	~**x** = -11 (1111 0101)
		^	Bitwise XOR	**x** ^ **y** = 14 (0000 1110)
		>>	Bitwise right shift	**x** » **2** = 2 (0000 0010)
		<<	Bitwise left shift	**x** « **2** = 40 (0010 1000)

[23]:

```python
a = 2 #binary: 0010
b = 3 #binary: 0011
print('a & b =',a & b,"=",bin(a&b))
```

```
a & b = 2 = 0b10
```

[24]:

```python
5 >> 1

# 0   0000 0101
#       0000 0010
# 0010 is 2 in decimal
```

[24]:

```
2
```

Explanation:

0000 0101 -> 5 (5 is 0101 in binary)

Shifting the digits by 1 to the right and zero padding that will be: 0   0000 0101 = 0000 0010

0000 0010 -> 2

[25]:

```python
5 << 1

# 0000 0101    0
# 0000 1010
# 1010 is 10 in decimal
```

[25]:

```
10
```

Explanation:

0000 0101 -> 5

Shifting the digits by 1 to the left and zero padding will be: 0000 0101   0 = 0000 1010

0000 1010 -> 10

[26]:
```
6 >> 2  # What will be output???
```

[26]:
```
1
```

[27]:
```
print(not (True and False), "==", not True or not False)
#       TRUE                       ==      True
```

```
True == True
```

[28]:
```
print (False and (not False) or (False and True), "==",not (True and (not␣
 ↪False) or (not True)))
```

```
False == False
```

1.5 5. Assignment operators

Assignment operators are used in Python to **assign values** to **variables**.

`a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable `a` on the left.

There are various compound operators in Python like a `+= 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Symbol	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
\\|=	x |= 5	x = x | 5
^=	x ˆ= 5	x = x ˆ 5
>>=	x »= 5	x = x » 5
<<=	x «= 5	x = x « 5

The binary operators can be combined with assignment to modify a variable value. For example:

[29]:
```
x = 1
x += 2 # add 2 to x
print("x is",x)
x <<= 2 # left shift by 2 (equivalent to x *= 4)
print('x is',x)
x **= 2 # x := x^2
```

```
print('x is',x)
```

```
x is 3
x is 12
x is 144
```

1.6 6. Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

1.6.1 1. Identity operators

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the **memory**. Two variables that are equal does not imply that they are **identical**.

Symbol	Meaning	Example
`is`	True if the operands are identical (refer to the same object)	**x is True**
`is not`	True if the operands are not identical (do not refer to the same object)	**x is not True**

Example : Identity operators in Python

[30]:

```
x1 = 6
y1 = 6
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3] # list
y3 = [1,2,3] # list

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False because two list [] can never be equal
print(x3 is y3)
```

```
False
True
False
```

Explanation:

Here, we see that **x1** and **y1** are integers of same values, so they are equal as well as identical. Same is the case with **x2** and **y2** (strings).

But **x3** and **y3** are list. They are equal but not identical. It is because interpreter locates them **separately in memory** although they are equal.

1.6.2 2. Membership operators

in and **not in** are the membership operators in Python. They are used to test whether a value or variable is found in a **sequence** (**string**, **list**, **tuple**, **set** and **dictionary**).

In a dictionary we can only test for presence of **key, not the value**.

Symbol	Meaning	Example
in	True if value/variable is found in sequence	**5 in x**
not in	True if value/variable is not found in sequence	**5 not in x**

Example : Membership operators in Python

[31]:
```
x = 'Hello world'
y = {1:'a',2:'b'} # dictionary 1 is key and 'a' is element. So we access␣
 ↪element without its key.

# Output: True
print('H' in x)  # Do we have 'H' in 'Hello World' ?

# Output: True
print('hello' not in x)  # Do we have 'hello' in 'Hello World' ?

# Output: True
print(1 in y)

# Output: False because we cannot identify 'a' without its key hence it is␣
 ↪Flase.
print('a' in y)
```

```
True
True
True
False
```

Explanation:

Here, **'H'** is in **x** but **'hello'** is not present in **x** (remember, Python is case sensitive). Similary, **1** is key and **'a'** is the value in dictionary y. Hence, **'a'in y** returns **False**.

1.7 Exercises Operators

1. Declare your age as integer variable
2. Declare your height as a float variable
3. Declare a variable that store a complex number
4. Write a code that prompts the user to enter base and height of the triangle and calculate an area of this triangle (area = 0.5 x b x h).

```
Enter base: 20
Enter height: 10
The area of the triangle is 100
```

5. Write a code that prompts the user to enter side a, side b, and side c of the triangle. Calculate the perimeter of the triangle (perimeter = a + b + c).

```
Enter side a: 5
Enter side b: 4
Enter side c: 3
The perimeter of the triangle is 12
```

6. Get length and width of a rectangle using prompt. Calculate its area (**area = length x width**) and perimeter (**perimeter = 2 x (length + width)**)
7. Get radius of a circle using prompt. Calculate the area (**area = pi x r x r**) and circumference (**c = 2 x pi x r**) where pi = 3.14.
8. Calculate the slope, x-intercept and y-intercept of $y = 2x - 2$
9. Slope is ($m = (y2 - y1)/(x2 - x1)$). Find the slope and **Euclidean distance** between point (2, 2) and point (6,10)
10. Compare the slopes in tasks 8 and 9.
11. Calculate the value of y ($y = x^2 + 6x + 9$). Try to use different x values and figure out at what x value y is going to be 0.
12. Find the length of **'python'** and **'datascience'** and make a falsy comparison statement.
13. Use **and** operator to check if **on** is found in both **python** and **cannon**
14. **I hope this course is not full of jargon**. Use **in** operator to check if **jargon** is in the sentence.
15. There is no **on** in both **python** and **cannon**
16. Find the length of the text **python** and convert the value to float and convert it to string
17. Even numbers are divisible by 2 and the remainder is zero. How do you check if a number is even or not using python?
18. Check if the floor division of 7 by 3 is equal to the int converted value of 2.7.
19. Check if type of **“10”** is equal to type of 10
20. Check if int(**“9.6”**) is equal to 10
21. Write a code that prompts the user to enter hours and rate per hour. Calculate pay of the person?

```
Enter hours: 40
Enter rate per hour: 30
Your weekly earning is 1200
```

22. Write a script that prompts the user to enter number of years. Calculate the number of seconds a person can live. Assume a person can live hundred years

```
Enter number of years you have lived: 100
You have lived for 3153600000 seconds.
```

23. Write a Python code that displays the following table

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

```
[ ]:
```

Python_List_Comprehension

March 2, 2023

1 Python List Comprehension

In this class, we will learn about Python list comprehensions, and how to use it.

List comprehension is an elegant and concise way to create a new list from an existing list in Python. It is a short way to create a new list. List comprehension is considerably faster than processing a list using the `for loop`.

1.1 List Comprehension vs String vs For Loop in Python

Suppose, we want to separate the letters of the word **`python`** and add the letters as items of a list.

The first thing that comes in mind would be using **String**.

[1]:
```python
# Example 1: Converting string to a list
# Method 1:

language = 'python'
lst = list(language) # changing the string to list

print(lst)            # ['p', 'y', 't', 'h', 'o', 'n']
print(type(lst))      # list
```

```
['p', 'y', 't', 'h', 'o', 'n']
<class 'list'>
```

The second thing that comes in mind would be using **for loop**.

[2]:
```python
# Example 1: Iterating through a string Using for Loop
# Method 2:

p_letters = []

for letter in 'python':
    p_letters.append(letter)

print(p_letters)       # ['p', 'y', 't', 'h', 'o', 'n']
print(type(p_letters)) # List
```

```
['p', 'y', 't', 'h', 'o', 'n']
<class 'list'>
```

Explanation:

However, Python has an easier way to solve this issue using List Comprehension. List comprehension is an elegant way to define and create lists based on existing lists.

Let's see how the above program can be written using **list comprehension**.

[3]:

```python
# Example 1: Iterating through a string Using List Comprehension
# Method 3:

p_letters = [ letter for letter in 'python' ]
print(p_letters)       # ['p', 'y', 't', 'h', 'o', 'n']
print(type(p_letters)) # List
```

```
['p', 'y', 't', 'h', 'o', 'n']
<class 'list'>
```

Explanation:

In the above example, a new list is assigned to variable **`p_letters`**, and list contains the items of the iterable string 'human'. We call **`print()`** function to receive the output.

1.2 Syntax of List Comprehension

```
[expression for item in list]
```

We can now identify where list comprehensions are used.

If you noticed, **`python`** is a string, not a list. This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a **list**.

You can do that using loops. However, not every loop can be rewritten as list comprehension. But as you learn and get comfortable with list comprehensions, you will find yourself replacing more and more loops with this elegant syntax.

1.3 Conditionals in List Comprehension

List comprehensions can utilize conditional statement to modify existing list (or other tuples). We will create list that uses mathematical operators, integers, and **range()**.

[4]:

```python
# Example 4: regular program to generate power for the numbers

pow2 = [2 ** x for x in range(10)]
print(pow2)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Above code is equivalent to:

[5]:
```
# Example 4: with list comprehension

pow2 = []
for x in range(10):
    pow2.append(2 ** x)
print(pow2)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

A list comprehension can optionally contain more `for` or if statements. An optional `if` statement can filter out items for the new list. Here are some examples.

1.3.1 For instance if you want to generate a list of numbers

[6]:
```
# Example 2:
# Method 1: Generating numbers

numbers = [i for i in range(11)]  # to generate numbers from 0 to 10
print(numbers)                    # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[7]:
```
# Example 2:
# Method 2: It is possible to do mathematical operations during iteration

squares = [i * i for i in range(11)]
print(squares)                    # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

[8]:
```
# Example 2:
# Method 3: It is also possible to make a list of tuples

numbers = [(i, i * i) for i in range(11)]
print(numbers)                            # [(0, 0), (1, 1), (2, 4), (3, 9),␣
 ↪(4, 16), (5, 25)]
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64),
(9, 81), (10, 100)]
```

[9]:
```
# Example 4:

pow2 = [2 ** x for x in range(10) if x > 5]
pow2
```

[9]:
```
[64, 128, 256, 512]
```

1.3.2 List comprehension can be combined with if statement

[10]:

```
# Example 7: Using 'if' with List Comprehension

number_list = [ x for x in range(30) if x % 2 == 0]
print(number_list)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Explanation:

The list, **number_list**, will be populated by the items in range from 0-27 if the item's value is divisible by 2.

[11]:

```
# Generating even numbers

even_numbers = [x for x in range(30) if x % 2 == 0]  # to generate even numbers␣
  ↪list in range 0 to 21
print(even_numbers)                                   # [0, 2, 4, 6, 8, 10, 12,␣
  ↪14, 16, 18, 20]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

[12]:

```
# Generating odd numbers

odd_numbers = [x for x in range(21) if x % 2 != 0]  # to generate odd numbers␣
  ↪in range 0 to 21
# odd_numbers = [x for x in range(21) x % 2 == 1]   # this would also work!
print(odd_numbers)                                   # [1, 3, 5, 7, 9, 11, 13,␣
  ↪15, 17, 19]
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

[13]:

```
# Filter numbers: let's filter out positive even numbers from the list below

numbers = [-8, -7, -3, -1, 0, 1, 3, 4, 5, 7, 6, 8, 10]
positive_even_numbers = [x for x in range(21) if x % 2 == 0 and x > 0]
print(positive_even_numbers)                         # [2, 4, 6, 8, 10, 12, 14,␣
  ↪16, 18, 20]
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

[14]:

```
# Flattening a three dimensional array

list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_list = [number for row in list_of_lists for number in row]
print(flattened_list)                               # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

[15]:

```
# Example 9:

[x+y for x in ['Python ','C '] for y in ['Language','Programming']]
```

[15]:

```
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

[16]:

```
# Example 10: Nested 'if' with List Comprehension

num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
print(num_list)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Explanation:

Here, list comprehension checks:

- Is `y` divisible by 2 or not?
- Is `y` divisible by 5 or not?

If `y` satisfies both conditions, `y` is appended to `num_list`.

[17]:

```
# Example 11: if...else With List Comprehension

obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
```

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

Explanation:

Here, list comprehension will check the 10 numbers from 0 to 9. If `i` is divisible by 2, then `Even` is appended to the `obj` list. If not, `Odd` is appended.

1.4 List Comprehensions vs Lambda functions

List comprehensions aren't the only way to work on lists. Various built-in functions and lambda functions can create and modify lists in less lines of code.

1.4.1 Lambda Function

Lambda function is a small anonymous function without a name. It can take any number of arguments, but can only have one expression. Lambda function is similar to anonymous functions in JavaScript. We need it when we want to write an anonymous function inside another function.

1.4.2 Creating a Lambda Function

To create a lambda function we use `lambda` keyword followed by a parameter(s), followed by an expression. See the syntax and the example below. Lambda function does not use return but it explicitly returns the expression.

Syntax:

```
x = lambda param1, param2, param3: param1 + param2 + param2
print(x(arg1, arg2, arg3))
```

[18]:

```
# Named function
def add_two_nums(a, b):
    return a + b

print(add_two_nums(2, 3))      # 5
```

```
5
```

Lets change the above function to a lambda function

[19]:

```
add_two_nums = lambda a, b: a + b
print(add_two_nums(2,3))     # 5
```

```
5
```

[20]:

```
# Self invoking lambda function
(lambda a, b: a + b)(2,3) # 5
```

[20]:

```
5
```

[21]:

```
square = lambda x : x ** 2
print(square(3))     # 9
cube = lambda x : x ** 3
print(cube(3))     # 27
```

```
9
27
```

[22]:

```
# Multiple variables

multiple_variable = lambda a, b, c: a ** 2 - 3 * b + 4 * c
print(multiple_variable(5, 5, 3)) # 22
```

```
22
```

[23]:

```
### Lambda Function Inside Another Function

def power(x):
    return lambda n : x ** n

cube = power(2)(3)   # function power now need 2 arguments to run, in separate rounded brackets
print(cube)          # 8
two_power_of_five = power(2)(4)
print(two_power_of_five)  # 16
```

```
8
16
```

[24]:

```
# Example 3: Using Lambda functions inside List

letters = list(map(lambda x: x, 'python'))
print(letters)
```

```
['p', 'y', 't', 'h', 'o', 'n']
```

Explanation:

However, list comprehensions are usually more human readable than lambda functions. It is easier to understand what the programmer was trying to accomplish when list comprehensions are used.

1.5 Nested Loops in List Comprehension

Suppose, we need to compute the transpose of a matrix that requires nested for loop. Let's see how it is done using normal for loop first.

[25]:

```
# Example 12: Transpose of Matrix using Nested Loops

transposed = []
matrix = [[1, 2, 3, 4], [5, 6, 8, 9]]

for i in range(len(matrix[0])):
    transposed_row = []

    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)

print(transposed)
```

```
[[1, 5], [2, 6], [3, 8], [4, 9]]
```

Explanation:

The above code use two for loops to find transpose of the matrix.

We can also perform nested iteration inside a list comprehension. In this section, we will find transpose of a matrix using nested loop inside list comprehension.

[26]:

```
# Example 13: Transpose of a Matrix using List Comprehension

matrix = [[1,2], [3,4], [5,6], [7,8]]
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
```

```
[[1, 3, 5, 7], [2, 4, 6, 8]]
```

Explanation:

In above program, we have a variable **`matrix`** which have **`4`** rows and **`2`** columns. We need to find transpose of the **`matrix`**. For that, we used list comprehension.

> **Note:** The nested loops in list comprehension don't work like normal nested loops. In the above program, **`for i in range(2)`** is executed before **`row[i] for row in matrix`**. Hence at first, a value is assigned to **`i`** then item directed by **`row[i]`** is appended in the **`transpose`** variable.

1.6 Key Points to Remember

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.

1.7 Exercises List Comprehension

1. Filter only negative and zero in the list using list comprehension

   - `numbers = [-4, -3, -2, -1, 0, 3, 6, 9, 12]`

2. Flatten the following list of lists of lists to a one dimensional list :

   - “‘py list_of_lists =[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]]]

output: [1, 2, 3, 4, 5, 6, 7, 8, 9] “‘

3. Using list comprehension create the following list of tuples:

   -
     ```
     [(0, 1, 0, 0, 0, 0, 0),
     (1, 1, 1, 1, 1, 1, 1),
     (2, 1, 2, 4, 8, 16, 32),
     (3, 1, 3, 9, 27, 81, 243),
     (4, 1, 4, 16, 64, 256, 1024),
     (5, 1, 5, 25, 125, 625, 3125),
     (6, 1, 6, 36, 216, 1296, 7776),
     (7, 1, 7, 49, 343, 2401, 16807),
     (8, 1, 8, 64, 512, 4096, 32768),
     (9, 1, 9, 81, 729, 6561, 59049),
     (10, 1, 10, 100, 1000, 10000, 100000)]
     ```

4. Flatten the following list to a new list:

   -
     ```
     countries = [[('INDIA', 'MUMBAI')], [('CHINA', 'SHANGHAI')], [('FINLAND', 'TAMPERE')]]
     output:
     [['INDIA','IN', 'MUMBAI'], ['CHINA', 'CH', 'SHANGHAI'], ['FINLAND','FI', 'TAMPERE']]
     ```

5. Change the following list to a list of dictionaries:

   -
     ```
     countries = [[('India', 'Mumbai')], [('China', 'Shanghai')], [('Finland', 'Tampere')]]
     output:
     ```

```
[{'country': 'INDIA', 'city': 'MUMBAI'},
{'country': 'CHINA', 'city': 'SHANGHAI'},
{'country': 'FINLAND', 'city': 'TAMPERE'}]
```

6. Change the following list of lists to a list of concatenated strings:

   - `names = [[('Milaan', 'Parmar')], [('Arthur', 'Curry')], [('Bill', 'Gates')], [('Ethan`
     ```
     output
     ['Milaan Parmar', 'Arthur Curry', 'Bill Gates', 'Ethan Hunt']
     ```

7. Write a lambda function which can solve a slope or y-intercept of linear functions.

[ ]: