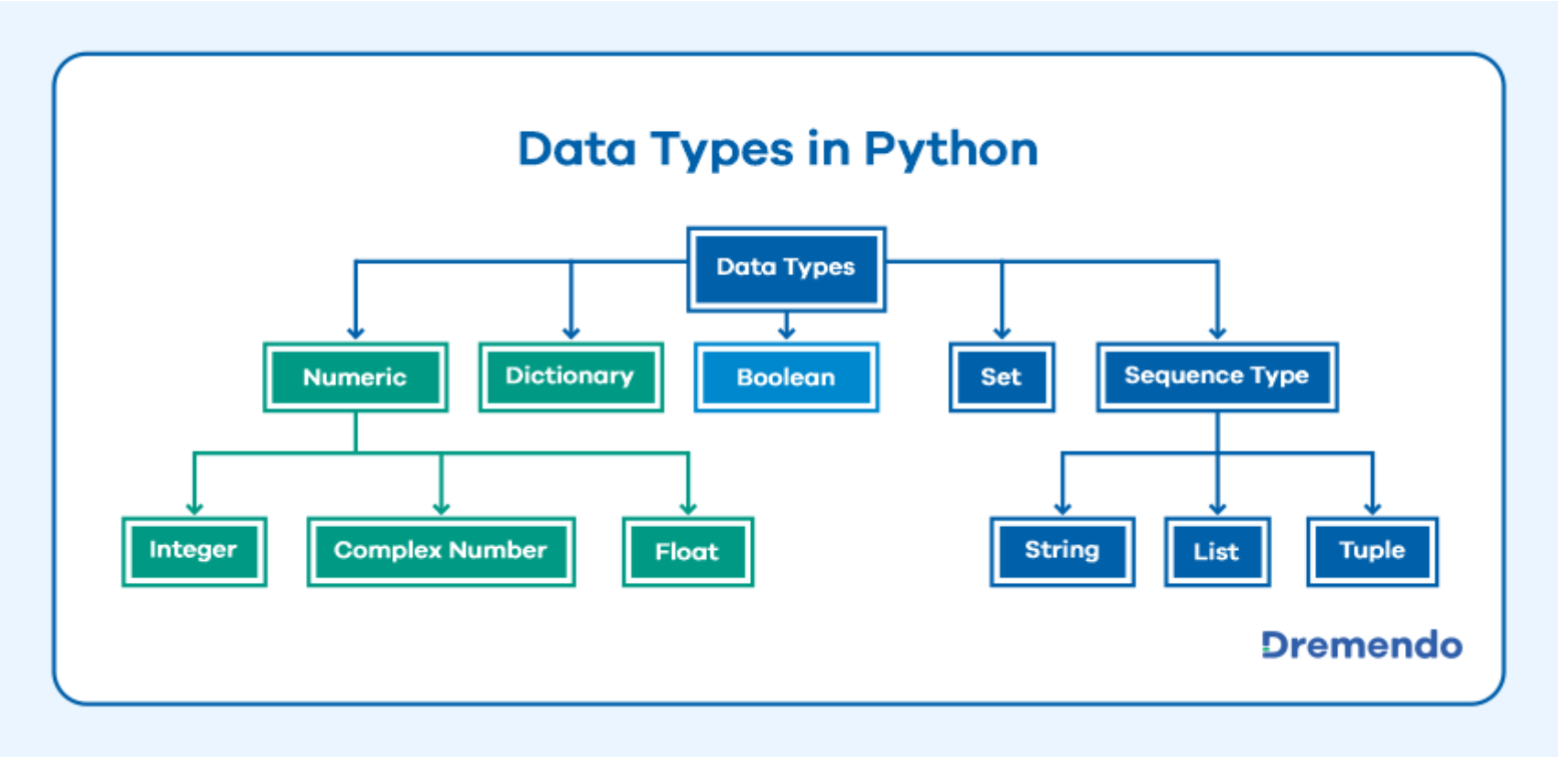
Python Data Types

Data types specify the different sizes and values that can be stored in the variable. For example, Python stores numbers, strings, and a list of values using different data types.

Python is a dynamically typed language; therefore, we do not need to specify the variable's type while declaring it. Whatever value we assign to the variable based on that data type will be automatically assigned. For example, name = 'Deepali' here Python will store the name variable as a str data type.

No matter what value is stored in a variable (object), a variable can be any type like int, float, str, list, set, tuple, dict, bool, etc.

- Numeric: int, float, and complex
- Sequence: String, list, and tuple
- Set
- Dictionary (dict)



To check the data type of variable use the built-in function type() and isinstance().

- The type() function returns the data type of the variable
- The isinstance() function checks whether an object belongs to a particular class.

Str data type

In Python, A string is a sequence of characters enclosed within a single quote or double quote. These characters could be anything like letters, numbers, or special symbols enclosed within double quotation marks. For example, "Deepali" is a string.

The string type in Python is represented using a str class.

To work with text or character data in Python, we use Strings. Once a string is created, we can do many operations on it, such as searching inside it, creating a substring from it, and splitting it.

In [36]:
```
  raw = r'this\t\n and that'

  # this\t\n and that
  print (raw)

  multi = """It was the best of times.
  It was the worst of times."""

  # It was the best of times.
  #   It was the worst of times.
print (multi)
```

```
this\t\n and that
It was the best of times.
It was the worst of times.
```

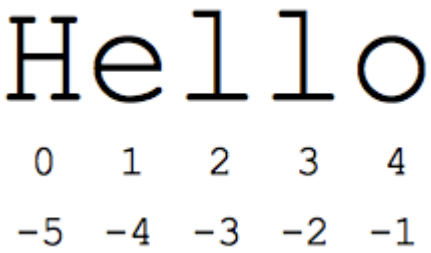
String Methods

Here are some of the most common string methods. A method is like a function, but it runs "on" an object. If the variable s is a string, then the code s.lower() runs the lower() method on that string object and returns the result (this idea of a method running on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Here are some of the most common string methods:

- s.lower(), s.upper() -- returns the lowercase or uppercase version of the string
- s.strip() -- returns a string with whitespace removed from the start and end
- s.isalpha()/s.isdigit()/s.isspace()... -- tests if all the string chars are in the various character classes
- s.startswith('other'), s.endswith('other') -- tests if the string starts or ends with the given other string
- s.find('other') -- searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found
- s.replace('old', 'new') -- returns a string where all occurrences of 'old' have been replaced by 'new'
- s.split('delim') -- returns a list of substrings separated by the given delimiter. The delimiter is not a regular expression, it's just text. 'aaa,bbb,ccc'.split(',') -> ['aaa', 'bbb', 'ccc']. As a convenient special case s.split() (with no arguments) splits on all whitespace chars.
- s.join(list) -- opposite of split(), joins the elements in the given list together using the string as the delimiter. e.g. '---'.join(['aaa', 'bbb', 'ccc']) -> aaa---bbb---ccc

String Slices

The "slice" syntax is a handy way to refer to sub-parts of sequences -- typically strings and lists. The slice s[start:end] is the elements beginning at start and extending up to but not including end. Suppose we have s = "Hello"



- s[1:4] is 'ell' -- chars starting at index 1 and extending up to but not including index 4
- s[1:] is 'ello' -- omitting either index defaults to the start or end of the string
- s[:] is 'Hello' -- omitting both always gives us a copy of the whole thing (this is the pythonic way to copy a sequence like a string or list)
- s[1:100] is 'ello' -- an index that is too big is truncated down to the string length

The standard zero-based index numbers give easy access to chars near the start of the string. As an alternative, Python uses negative numbers to give easy access to the chars at the end of the string: s[-1] is the last char 'o', s[-2] is 'l' the next-to-last char, and so on. Negative index numbers count back from the end of the string:

- s[-1] is 'o' -- last char (1st from the end)
- s[-4] is 'e' -- 4th from the end
- s[:-3] is 'He' -- going up to but not including the last 3 chars.
- s[-3:] is 'llo' -- starting with the 3rd char from the end and extending to the end of the string.
- It is a neat truism of slices that for any index n, s[:n] + s[n:] == s. This works even for n negative or out of bounds. Or put another way s[:n] and s[n:] always partition the string into two string parts, conserving all the characters.

In [1]:
```
name = "Deepali"
print(type(name))

# display "name" str
print(name)

#accessing 3rd charcter of the str
print(name[3])
```

```
<class 'str'>
Deepali
p
```

Note: The string is immutable, i.e., it can not be changed once defined. You need to create a copy of it if you want to modify it. This non-changeable behavior is called immutability.

In [3]:
```
# lets try to change 4th charcter of "name" str

name = "Deepali"

name[4] = "r"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-05b122a525da> in <module>
      3 name = "Deepali"
      4
----> 5 name[4] = "r"

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

Int data type

Python uses the int data type to represent whole integer values. For example, we can use the int data type to store the roll number of a student. The Integer type in Python is represented using a int class.

You can store positive and negative integer numbers of any length such as 235, -758, 235689741.

We can create an integer variable using the two ways

- Directly assigning an integer value to a variable
- Using a int() class.

In [4]:
```
# store int value
roll_no = 33
# display roll no
print("Roll number is:", roll_no)
# output 33
print(type(roll_no))
# output class 'int'

# store integer using int() class
id = int(25)
print(id)  # 25
print(type(id))  # class 'int'
```

```
Roll number is: 33
<class 'int'>
25
<class 'int'>
```

Float data type

To represent floating-point values or decimal values, we can use the float data type. For example, if we want to store the salary, we can use the float type.

The float type in Python is represented using a float class.

We can create a float variable using the two ways

- Directly assigning a float value to a variable
- Using a float() class.

In [5]:
```
# store a floating-point value
emp_salary = 8000.456
print("Salary is :", emp_salary)  # 8000.456
print(type(emp_salary))  # class 'float'

# store a floating-point value using float() class
num = float(54.75)
print(num)  # 54.75
print(type(num))  # class 'float'
```

```
Salary is : 8000.456
<class 'float'>
54.75
<class 'float'>
```

Complex data type

A complex number is a number with a real and an imaginary component represented as a+bj where a and b contain integers or floating-point values.

The complex type is generally used in scientific applications and electrical engineering applications. If we want to declare a complex value, then we can use the a+bj form. See the following example.

In [6]:
```
x = 9 + 8j  # both value are int type
y = 10 + 4.5j  # one int and one float
z = 11.2 + 1.2j  # both value are float type
print(type(x))  # class 'complex'>

print(x)  # (9+8j)
print(y)  # (10+4.5j)
print(z)  # (11.2+1.2j)
```

```
<class 'complex'>
(9+8j)
(10+4.5j)
(11.2+1.2j)
```

The real part of the complex number is represented using an integer value. The integer value can be in the form of either decimal, float, binary, or hexadecimal. But the imaginary part should be represented using the decimal form only. If we are trying to represent an imaginary part as binary, hex, or octal, we will get an error.

List data type

The Python List is an ordered collection (also known as a sequence) of elements. List elements can be accessed, iterated, and removed according to the order they inserted at the creation time.

We use the list data type to represent groups of the element as a single entity. For example: If we want to store all student's names, we can use list type.

The list can contain data of all data types such as int, float, string

Duplicates elements are allowed in the list

The list is mutable which means we can modify the value of list elements

We can create a list using the two ways

- By enclosing elements in the square brackets [].
- Using a list() class.

In [7]:
```
A = [ ] # This is a blank list variable
B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.
C = [2, 4, 'john'] # lists can contain different variable types.
```

All lists in Python are zero-based indexed. When referencing a member or the length of a list the number of list elements is always the number shown plus one.

In [22]:
```
mylist =  [ "john", "Deepali", " Raj " , "Neelam"]

b = len(mylist) ## This will return the length of the list which is 4. The index is 0, 1, 2, 3,4.

print("total element in mylist are : ",b)

print(mylist[1])     ## This will return the value at index 1, which is "Deepali"
print(mylist[0:2])  ## This will return the first 2 elements in the list.
```

```
total element in mylist are :  4
Deepali
['john', 'Deepali']
```

You can assign data to a specific element of the list using an index into the list. The list index starts at zero. Data can be assigned to the elements of an array as follows:

In [24]:
```
mylist = [0,1,2,3,4]
mylist[0] = 'John'
mylist[1] = 'Deepali'
mylist[2] = 'Raj'
mylist[3] = 'Neelam'
print( mylist[1])
```

```
Deepali
```

List Methods

Here are some other common list methods.

- list.append(elem) -- adds a single element to the end of the list. Common error: does not return the new list, just modifies the original.
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index. Throws a ValueError if the element does not appear (use "in" to check without a ValueError).
- list.remove(elem) -- searches for the first instance of the given element and removes it (throws ValueError if not present)
- list.sort() -- sorts the list in place (does not return it). (The sorted() function shown later is preferred.)
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Notice that these are *methods* on a list object, while len() is a function that takes the list (or string or whatever) as an argument.

In [2]:
```
mylist = ['Deepali', 'Rani', 'Harsh']
mylist.append('shemp')         ## append elem at end
mylist.insert(0, 'xxx')        ## insert elem at index 0
mylist.extend(['yyy', 'zzz'])  ## add list of elems at end
print (mylist)  ## ['xxx', 'Deepali', 'Rani', 'Harsh', 'shemp', 'yyy', 'zzz']
print (mylist.index('Rani'))    ## 2

mylist.remove('Rani')         ## search and remove that element
mylist.pop(1)                  ## removes and returns 'larry'
print (mylist)
```

```
['xxx', 'Deepali', 'Rani', 'Harsh', 'shemp', 'yyy', 'zzz']
2
['xxx', 'Harsh', 'shemp', 'yyy', 'zzz']
```

Common error: note that the above methods do not *return* the modified list, they just modify the original list.

In [31]:
```
mylist = [1, 2, 3]
print(mylist.append(4))   ## NO, does not work, append() returns None
  ## Correct pattern:
mylist.append(4)
print(mylist)  ## [1, 2, 3, 4]
```

```
None
[1, 2, 3, 4, 4]
```

List Build Up

One common pattern is to start a list a the empty list [], then use append() or extend() to add elements to it:

In [33]:
```
mylist = []          ## Start as the empty list
mylist.append('a')   ## Use append() to add elements
mylist.append('b')
```

List Slice

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

In [34]:
```
mylist = ['a', 'b', 'c', 'd']
print(mylist[1:-1])   ## ['b', 'c']
mylist[0:2] = 'z'    ## replace ['a', 'b'] with ['z']
print (mylist)        ## ['z', 'c', 'd']
```

```
['b', 'c']
['z', 'c', 'd']
```

Tuple data type

Tuples are ordered collections of elements that are unchangeable. The tuple is the same as the list, except the tuple is immutable means we can’t modify the tuple once created.

In other words, we can say a tuple is a read-only version of the list.

For example: If you want to store the roll numbers of students that you don’t change, you can use the tuple data type.

Note: Tuple maintains the insertion order and also, allows us to store duplicate elements.

We can create a tuple using the two ways

- 1. By enclosing elements in the parenthesis ()
- 2. Using a tuple() class.

In [37]:
```
# create a tuple
my_tuple = (11, 24, 56, 88, 78)
print(my_tuple)  # (11, 24, 56, 88, 78)
print(type(my_tuple))  # class 'tuple'

# Accessing 3rd element of a tuple
print(my_tuple[2])  # 56

# slice a tuple
print(my_tuple[2:7])  # (56, 88, 78)

# create a tuple using a tuple() class
my_tuple2 = tuple((10, 20, 30, 40))
print(my_tuple2)  # (10, 20, 30, 40)
```

```
(11, 24, 56, 88, 78)
<class 'tuple'>
56
(56, 88, 78)
(10, 20, 30, 40)
```

Tuple is immutable

A tuple is immutable means once we create a tuple, we can’t modify it

In [38]:
```
# create a tuple
my_tuple = (11, 24, 56, 88, 78)

# modify 2nd element of tuple
my_tuple[1] = 35
print(my_tuple)
# TypeError: 'tuple' object does not support item assignment
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-08e712c50398> in <module>
      3
      4 # modify 2nd element of tuple
----> 5 my_tuple[1] = 35
      6 print(my_tuple)
      7 # TypeError: 'tuple' object does not support item assignment

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

Dict data type

In Python, dictionaries are unordered collections of unique values stored in (Key-Value) pairs. Use a dictionary data type to store data as a key-value pair.

The dictionary type is represented using a dict class. For example, If you want to store the name and roll number of all students, then you can use the dict type.

In a dictionary, duplicate keys are not allowed, but the value can be duplicated. If we try to insert a value with a duplicate key, the old value will be replaced with the new value.

Dictionary has some characteristics which are listed below:

A heterogeneous (i.e., str, list, tuple) elements are allowed for both key and value in a dictionary. The dictionary is mutable which means we can modify its items Dictionary is unordered so we can’t perform indexing and slicing

We can create a dictionary using the two ways

- 1. By enclosing key and values in the curly brackets {}
- 2. Using a dict() class.

In [39]:
```
# create a dictionary
my_dict = {1: "Deepali", 2: "Neelam", 3: "Jaya"}

# display dictionary
print(my_dict)  # {1:"Deepali", 2: "Neelam", 3: "Jaya"}
print(type(my_dict))  # class 'dict'

# create a dictionary using a dit class
my_dict = dict({1: "Deepali", 2: "Neelam", 3: "Jaya"})

# display dictionary
print(my_dict)  # {1: "Deepali", 2: "Neelam", 3: "Jaya"}
print(type(my_dict))  # class 'dict'

# access value using a key name
print(my_dict[1])  # Deepali

# change the value of a key
my_dict[1] = "Kelly"
print(my_dict[1])  # Kelly
```

```
{1: 'Deepali', 2: 'Neelam', 3: 'Jaya'}
<class 'dict'>
{1: 'Deepali', 2: 'Neelam', 3: 'Jaya'}
<class 'dict'>
Deepali
Kelly
```

Set data type

In Python, a set is an unordered collection of data items that are unique. In other words, Python Set is a collection of elements (Or objects) that contains no duplicate elements.

In Python, the Set data type used to represent a group of unique elements as a single entity. For example, If we want to store student ID numbers, we can use the set data type.

The Set data type in Python is represented using a set class.

We can create a Set using the two ways

- By enclosing values in the curly brackets {}
- Using a set() class. The set data type has the following characteristics.
- It is mutable which means we can change set items
- Duplicate elements are not allowed
- Heterogeneous (values of all data types) elements are allowed
- Insertion order of elements is not preserved, so we can't perform indexing on a Set

In [3]:
```
# create a set using curly brackets{,}
my_set = {100, 25.75, "Deepali"}
print(my_set)  # {25.75, 100, 'Deepali'}
print(type(my_set))  # class 'set'

# create a set using set class
my_set = set({100, 25.75, "Deepali"})
print(my_set)  # {25.75, 100, 'Deepali'}
print(type(my_set))  # class 'set'

# add element to set
my_set.add(300)
print(my_set)  # {25.75, 100, 'Deepali', 300}

# remove element from set
my_set.remove(100)
print(my_set)  # {25.75, 'Deepali', 300}
```

```
{'Deepali', 25.75, 100}
<class 'set'>
{'Deepali', 25.75, 100}
<class 'set'>
{'Deepali', 25.75, 100, 300}
{'Deepali', 25.75, 300}
```

Bytes data type

In Python, to represent boolean values (True and False) we use the bool data type. Boolean values are used to evaluate the value of the expression. For example, when we compare two values, the expression is evaluated, and Python returns the boolean True or False.

In [41]:
```
x = 25
y = 20

z = x > y
print(z)  # True
print(type(z))  # class 'bool'
```

```
True
<class 'bool'>
```

Range data type

In Python, The built-in function range() used to generate a sequence of numbers from a start number up to the stop number. For example, If we want to represent the roll number from 1 to 20, we can use the range() type. By default, it returns an iterator object that we can iterate using a for loop.

In [14]:
```
# Generate integer numbers from 10 to 14
numbers = range(10,15)
print(type(numbers))  # class 'range'


```

```
<class 'range'>
```

In [13]:
```
x = range(10, 15, 1)
for n in x:
  print(n)
```

```
10
11
12
13
14
```