Python in Chemical Engineering: Basics and Applications

Mentor : HITESH BHATIA , hiteshtb21@iitk.ac.in

Software Required : Python 3.0

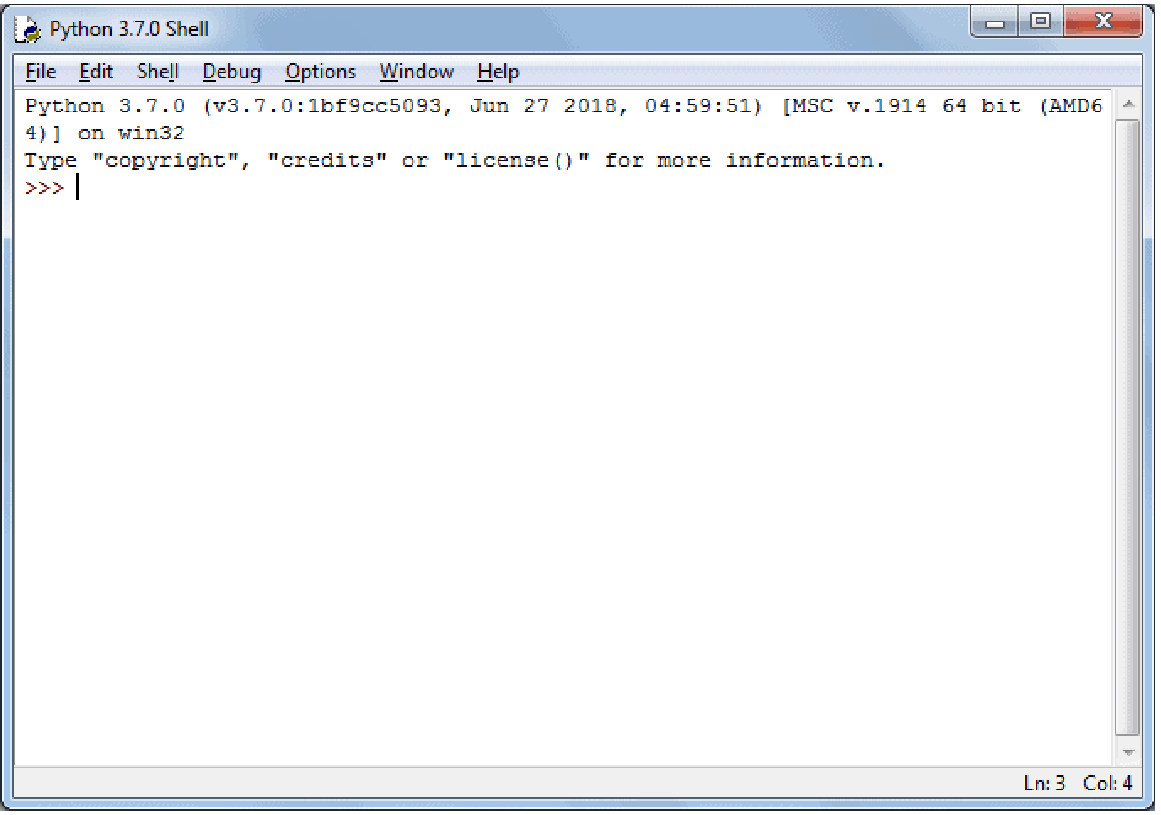
Pre-requisites : Material and Energy Balance Calculations ,Fluid Mechanics, Heat Transfer

Source Code → Python Interpreter → Output

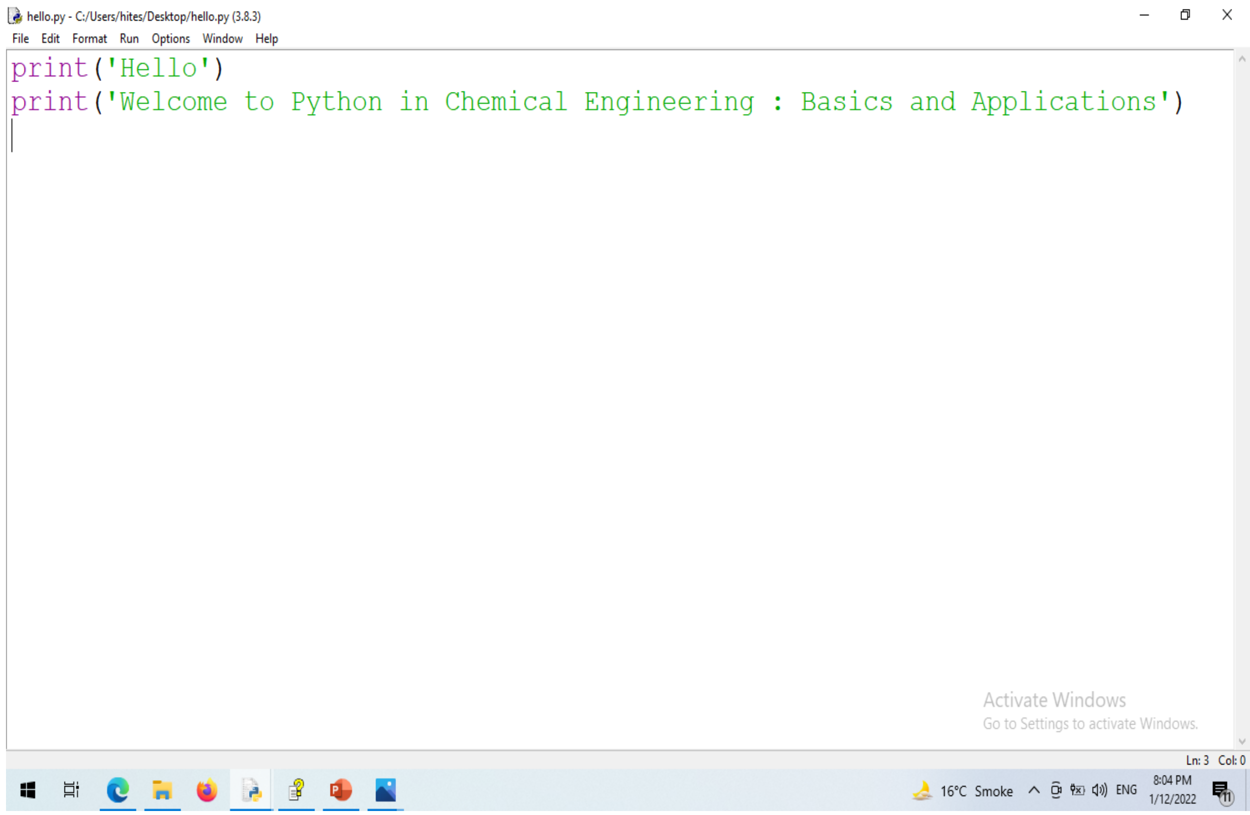
```
print(‘Hello World’)
```

A program that converts the code a developer writes into an intermediate language, called the byte code. It converts the code line by line, one at a time. It translates till the end and stops at that line where an error occurs, if any, making the debugging process easy.

Shell v/s IDLE



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

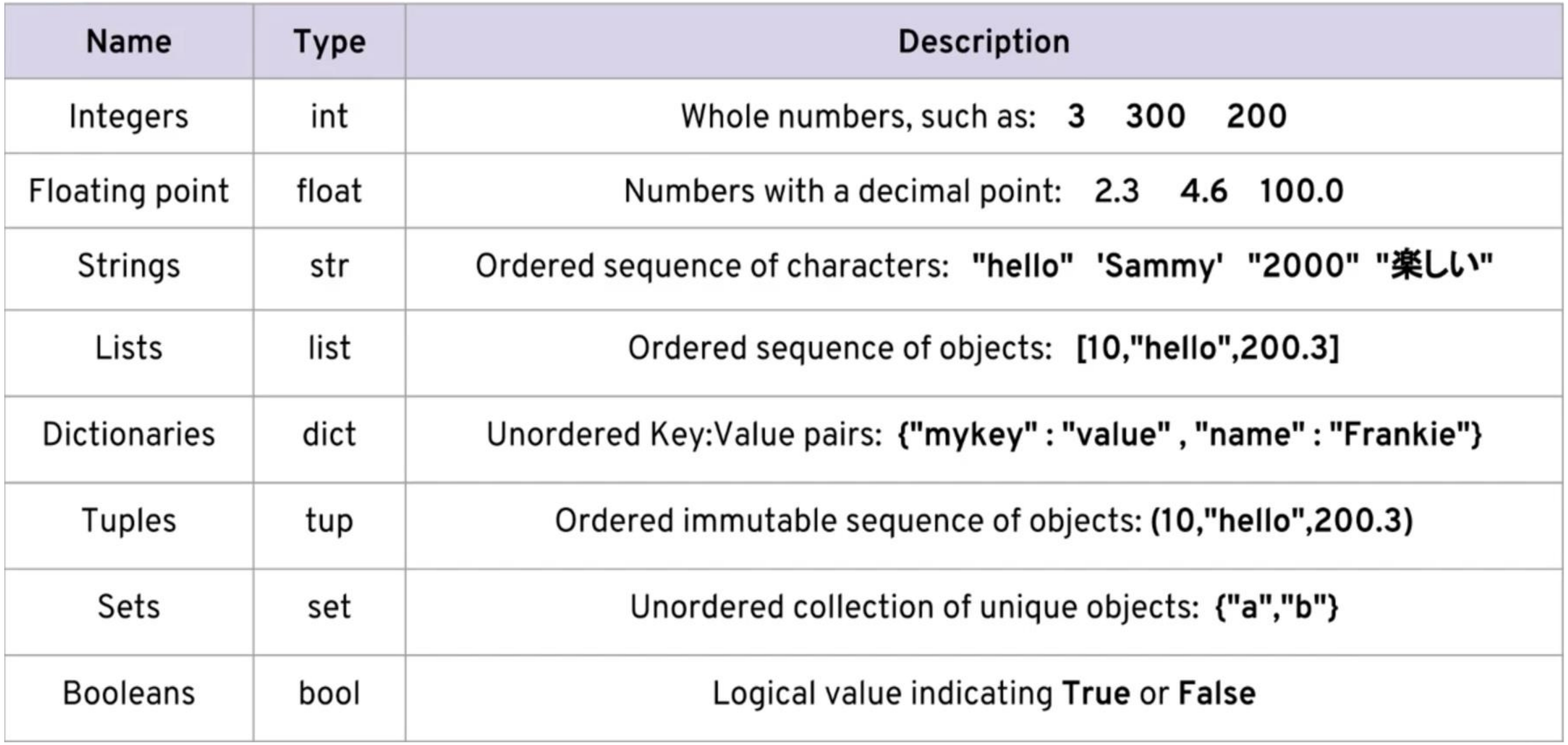

hello.py - C:/Users/hites/Desktop/hello.py (3.8.3)
File Edit Format Run Options Window Help
Activate Windows
Go to Settings to activate Windows.
Ln: 3 Col: 0
16°C Smoke
ENG
8:04 PM
1/12/2022

```
print('Hello')
print('Welcome to Python in Chemical Engineering : Basics and Applications')
```

We can use Python Interpreter in two modes:
1. Interactive Mode.
2. Script Mode.

- **In Interactive mode**, Python interpreter waits for you to enter command. When you type the command, Python interpreter goes ahead and executes the command, then it waits again for your next command.
- Python interpreter in interactive mode is commonly known as Python Shell.
- >>> is known as prompt string, it simply means that Python shell is ready to accept you commands. Python shell allows you type Python code and see the result immediately.
- Python Shell is great for testing small chunks of code but there is one problem - the statements you enter in the Python shell are not saved anywhere.
- In case, you want to execute same set of statements multiple times you would be better off to save the entire code in a file.
- Then, use the Python interpreter in script mode to execute the code from a file
- This mode of operation is **Script Mode.**

Expressions, Values and Data Types



Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

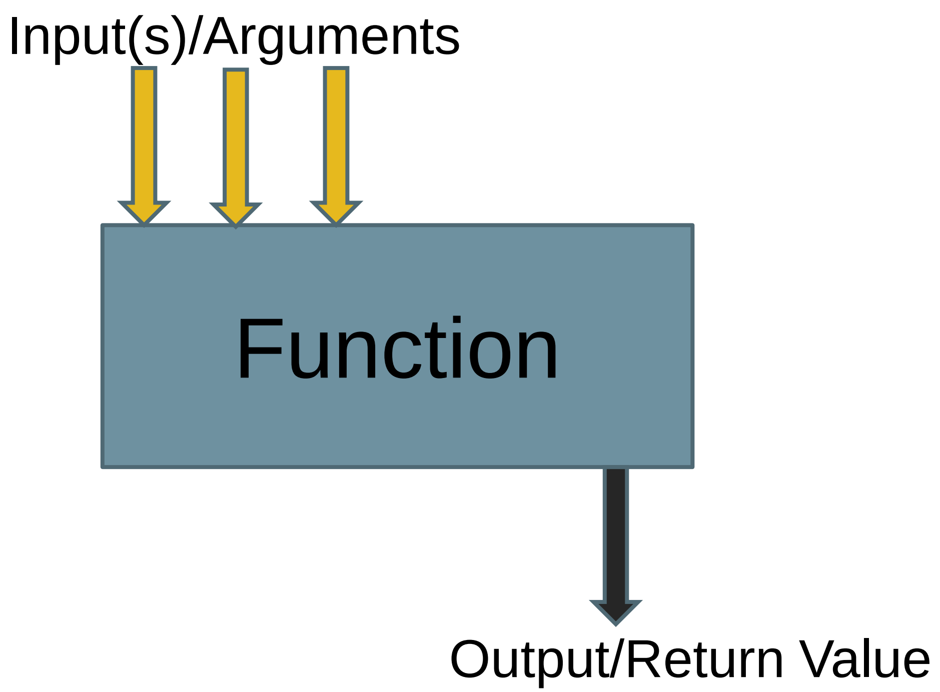
Operators and Operands

- You can build complex expressions out of simpler ones using **operators**. Operators are special tokens that represent computations like addition, multiplication and division. The values the operator works on are called **operands**.
- 20 + 32,5 ** 2,(5 + 9) * (15 - 7),print(7 + 5)
- The tokens +,- and * and the use of parentheses for grouping, mean in Python what they mean in mathematics. The asterisk (*) is the token for multiplication, and ** is the token for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

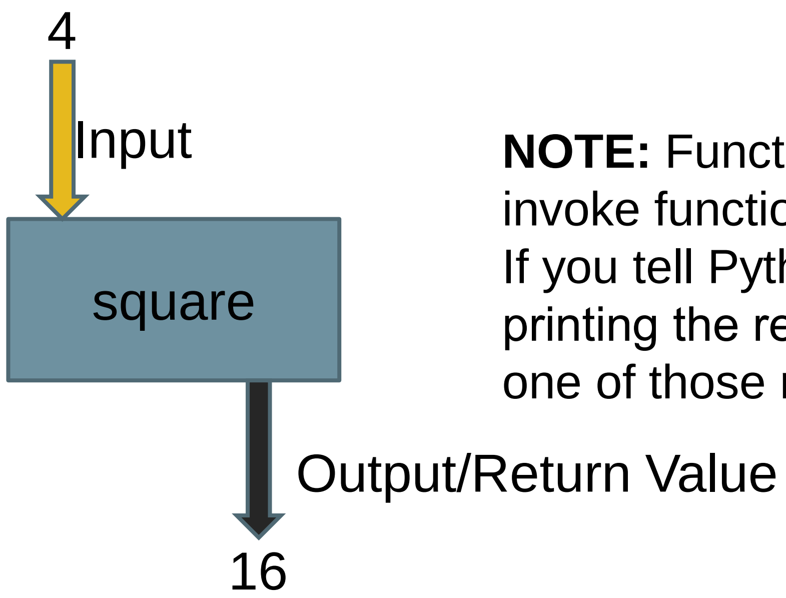
Operator Precedence

- When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.
  - *Parentheses* have the highest precedence and can be used to force an expression to evaluate in the order you want.
  - *Exponentiation* has the next highest precedence, so **2*1+1 is 3 and not 4** and **3*1**3 is 3 and not 27**.
  - *Multiplication and both division* operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence.
  - Operators with the *same* precedence are evaluated from left-to-right. In algebra we say they are *left-associative*. So in the expression 6-3+2 the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been 6-(3+2) which is 1.

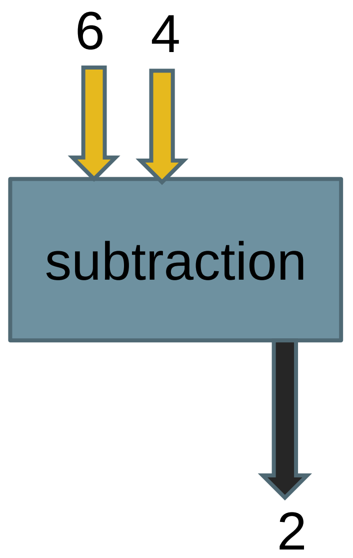
Function Calls



square(4)



sub(6,4)



NOTE: Functions are themselves just objects,parentheses invoke functions
If you tell Python to print the function object, rather than printing the results of invoking the function object, you'll get one of those not-so-nice printed representations.

Type Conversion Functions

int(a): This function converts **any data type to integer**.

float(‘100’): This function is used to convert **any data type to a** floating-point **number**

str(100.2) : converts the specified value into a string.

If you're not sure what type a given value has, then Python has a nice function called **type(object)**.
It returns the class type of the argument (object) passed as a parameter.

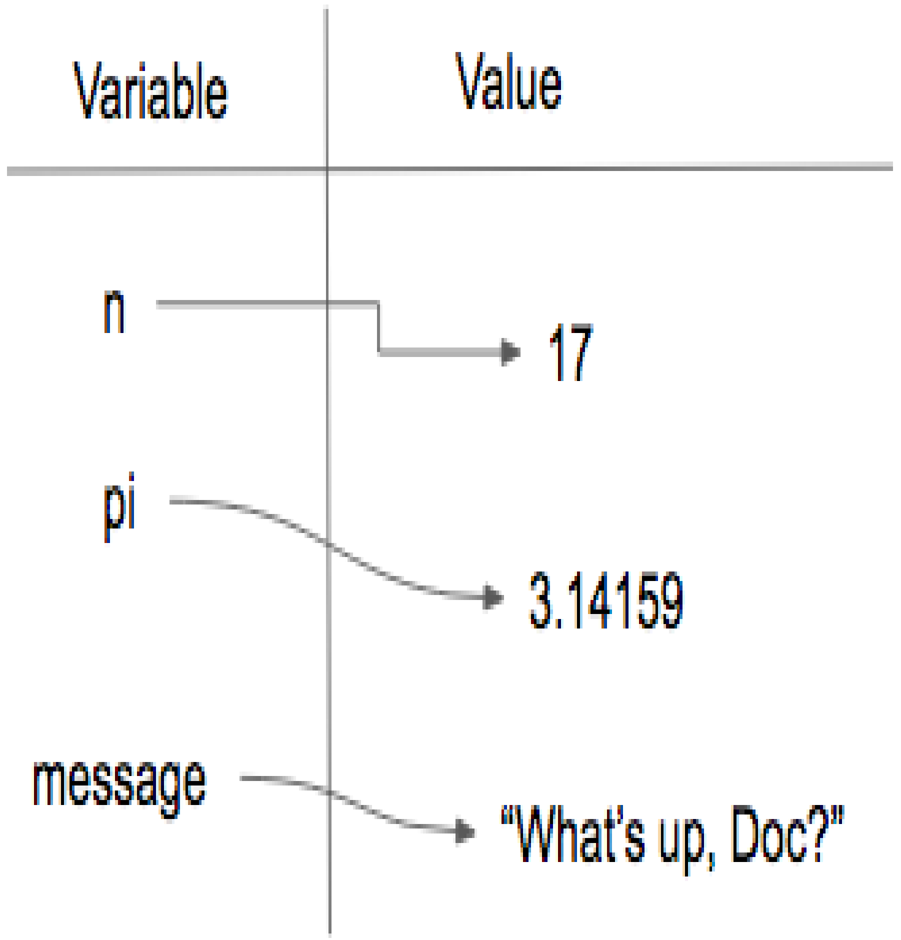
Variables and Assignment Operator

- A variable is a name that refers to a value.(Analogy of Π)
- **Assignment statements** create new variables and also give them values to refer to.

```
message='What's up !'
N=17
pi=3.14
```

- This example makes three assignments. The **assignment token**, **=** should not be confused with *equality* (we will see later that equality uses the (== token). Is 17=n a correct statement ?
- The assignment statement links a *name*, on the left hand side of the operator, with a *value*, on the right hand side.
- **TIP :** When reading or writing code, say to yourself "n is assigned 17" or "n gets the value 17" or "n is a reference to the object 17" or "n refers to the object 17". Don't say "n equals 17".

- This kind of figure, known as a **reference diagram**, is often called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable's state of mind). This diagram shows the result of executing the assignment statements shown.
- We use variables in a program to "remember" things, like the current score at the football game. But variables are *variable*.
- This means they can change over time, just like the scoreboard at a football game.
- You can assign a value to a variable, and later assign a different value to the same variable.
- **NOTE :** If you have programmed in another language such as Java or C++, you may be used to the idea that *variables* have types that are declared when the variable name is first introduced in a program. Python doesn't do that.
- Variables don't have types in Python; *values* do.



Updating Variable value

- One of the most common forms of reassignment is an **update** where the new value of the variable depends on the old. For example, **x=x+1**
- This means get the current value of x, add one, and then update x with the new value. The new value of x is the old value of x plus 1.
- If you try to update a variable that doesn’t exist, you get an error because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left.
- Before you can update a variable, you have to **initialize** it, usually with a simple assignment.
- **Short Hand notation** : In Python **+=** is used for incrementing and **-=** for decrementing.
- This means that x+=1 and x=x+1 have the same meaning.
- Can you guess what will be printed atlast?

```
x = 12
x = x - 3
x = x + 5
x = x + 1
print(x)
```

Taking Input from the user

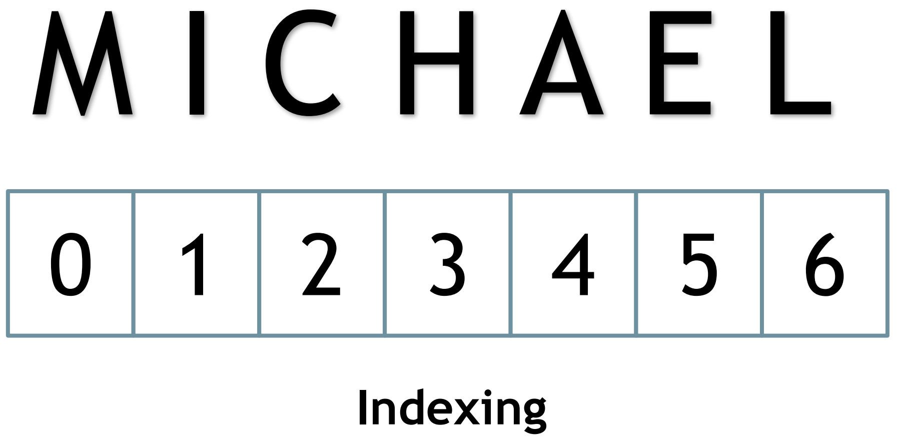
- Python has an inbuilt function to seek input from the user of the program.
- **n = input("Please enter your name: “)**
- The input function allows the programmer to provide a **prompt string**. In the example above, it is “Please enter your name: “When the function is evaluated, the prompt is shown.
- The user of the program can type some text and press ENTER/RETURN.
- When this happens the text that has been entered is returned from the **input** function and in this case assigned to the variable **n**.
- **NOTE - input** function returns a string value. Even if you asked the user to enter their age, you would get back a string like “17” . It would be your job, as the programmer, to convert that string into an int or a float, using the int or float converters we saw earlier.

Sequences : Strings and Lists

- Usually data is in the form of some kind of collection or sequence. Examples : Grocery list,Todo list,List of initial velocity components.
- So far we have seen built-in types like: **int,float** and **str. int** and **float.** are considered to be simple or primitive or atomic data types because their values are not composed of any smaller parts. They cannot be broken down.
- Strings and lists ,called **collection data types** are different from the others because they are made up of smaller pieces.
- In the case of strings, they are made up of smaller strings each containing one **character**.
- We will examine operations that can be performed on sequences, such as picking out individual elements or subsequences (called slices) or computing their length. In addition, we'll examine some special functions that are defined only for strings, and we'll find out one importance difference between strings and lists

Strings

- Strings can be defined as **immutable** sequential collections of characters.
- This means that the individual characters that make up a string are in a particular order from left to right.
- A string that contains no characters, often referred to as the **empty string**, is still considered to be a string.
- It is simply a sequence of zero characters and is represented by ‘’ or “” (two single or two double quotes with nothing in between).
- You can access a substring or part of a string using the indexing operator [].
- Indexing operator is handy in accessing a single character by its position or index value.
- Indexing in Python begins with 0.



Lists

- A **list** is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its **elements**.
- Similar to strings, except that the elements of a list can be of different types.
- The simplest is to enclose the elements in square brackets ([ ])
- my_list= [ ] → This creates an empty list
- my_list=[‘bag’,’shoes’,’bat’] (elements are of char type)
- This creates a list of three word elements.To create a list with elements inside it, put the list contents separated by comma.
- My_list=[2,5,’7’,’three’,5.55] (elements of this list are of mixed data type (int+char+float))

NOTE : When we say the first, third or nth character of a sequence, we generally mean counting the usual way, starting with 1. The nth character and the character AT INDEX n are different then: The nth character is at index n-1

```
note=[“hi”,2.5,5,[10,20]]
```

[“hi” , 2.5 , 5 ,[10,20]]

0	1	2	3

```
a=note[1]
print(a)

b=note[3][1]
print(b)

b=note[0][2]
print(b)
```

Disambiguating []: creation vs indexing

- Square brackets [] are used in quite a few ways in python.
- We have currently encountered two instances where we have used square brackets. The first is creating lists and the second is indexing.
- Indexing requires referencing an already created list while simply creating a list does not.

```
lst = [0]
n_lst = lst[0]
print(lst)
print(n_lst)
```

```
new_lst = ["NFLX", "AMZN", "GOOGL", "DIS", "XOM"]
part_of_new_lst = new_lst[0]
```

Note that in this example, what sets creating apart from indexing is the reference to the list to let python know that you are extracting an element from another list.

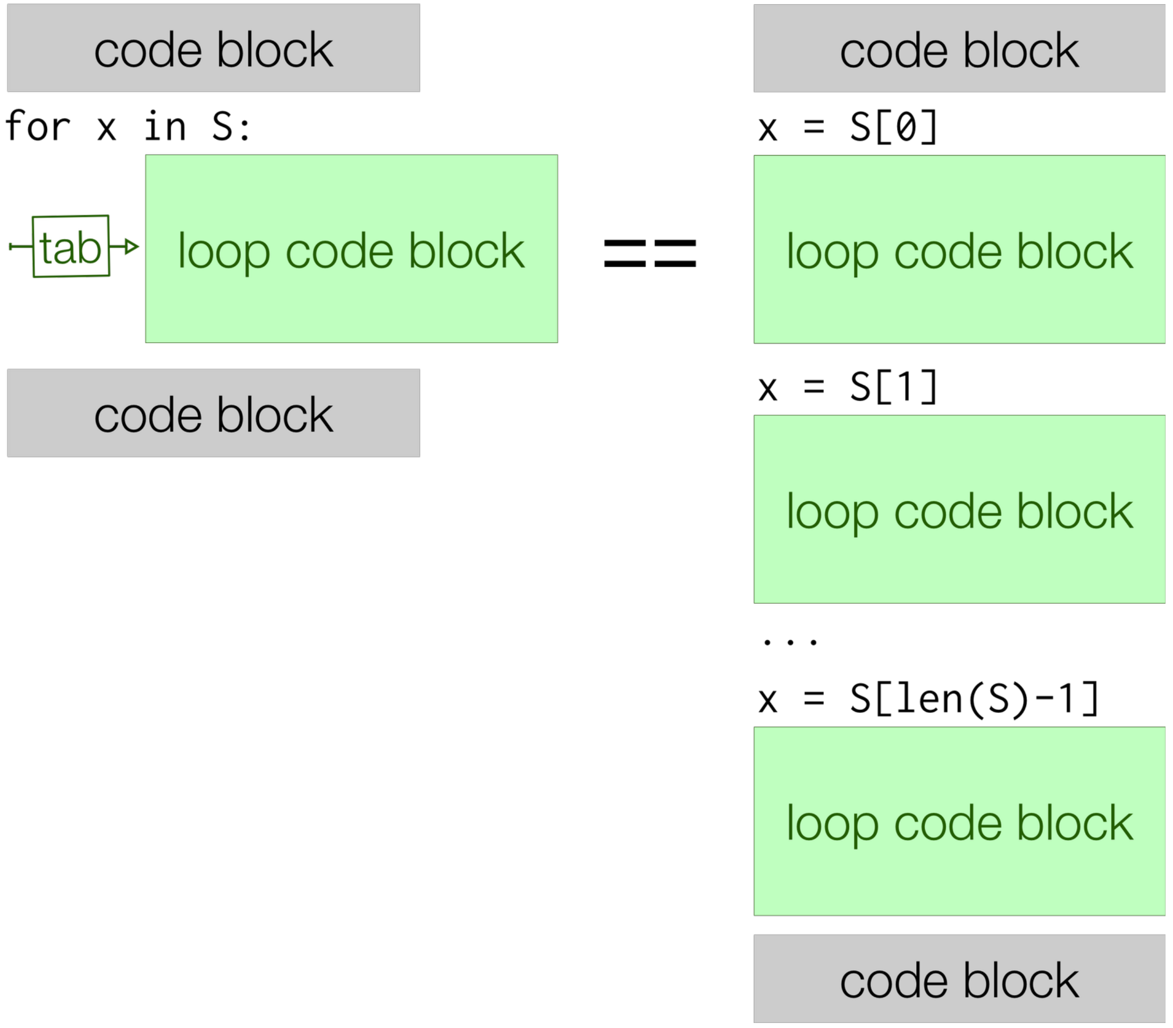
Iterations

- With collections (lists and strings), a lot of computations involve processing one item at a time.
- Often we start at the beginning, select each character in turn, do something to it, and continue until the end.
- For example, we could take each character and substitute for the character 13 characters away in the alphabet to create a coded message.

- The **for** loop :

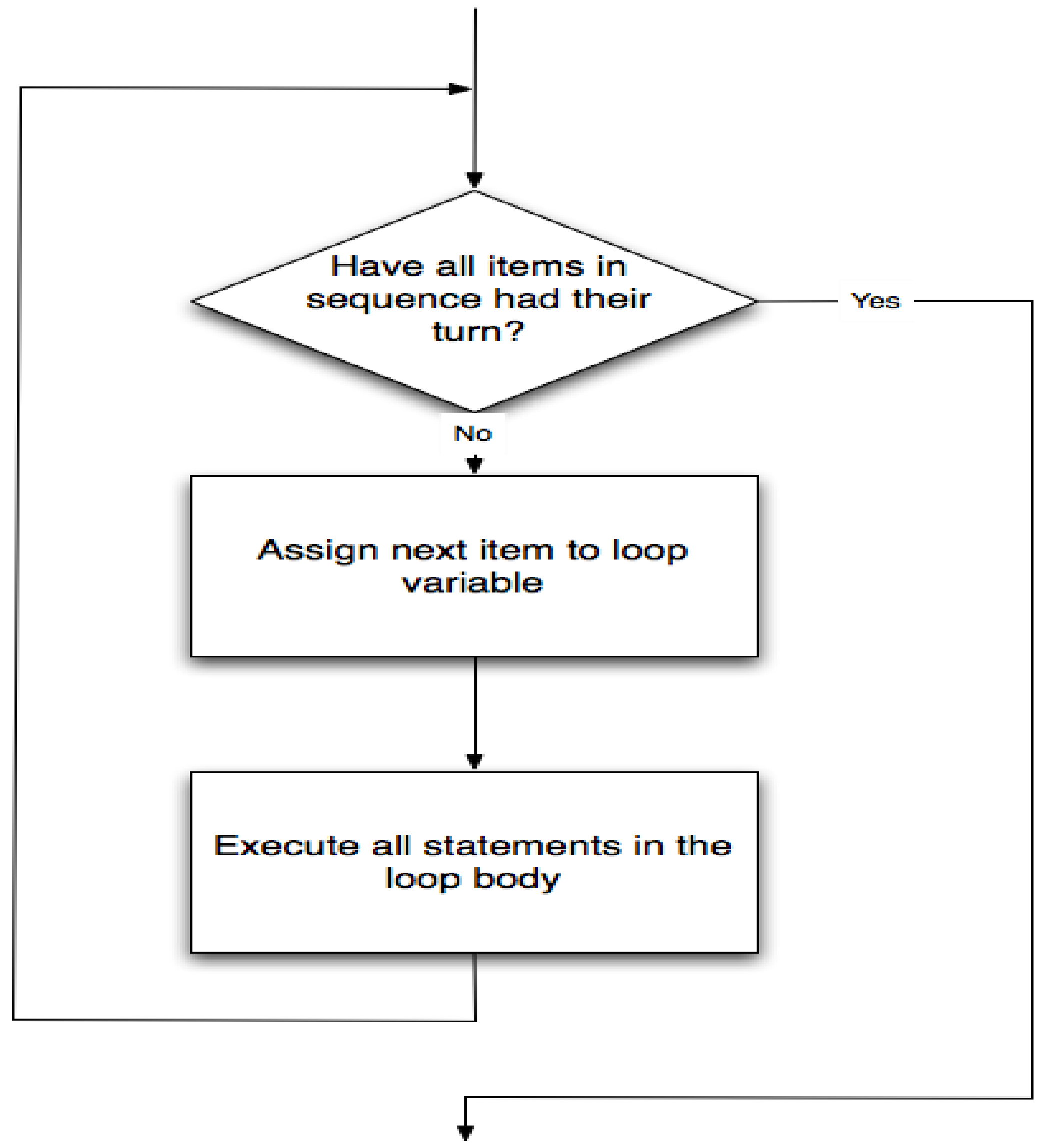
  Overall syntax

  for <loop_variable_name> in <sequence> :



Flow of execution of for loop

```
for name in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:
    print("Hi", name, "Please come to my party on Saturday!")
```



Have all items in sequence had their turn?
Yes
No
Assign next item to loop variable
Execute all statements in the loop body

Using the *range* Function to Generate a Sequence to Iterate Over

- The **range** function takes an integer n as input and returns a sequence of numbers, starting at 0 and going up to but not including n. Thus, instead of **range(3)** we can write **[0,1,2]**
- The **range** function takes at least one input - which should be an integer - and returns a list as long as your input.
- With one input, range will start at zero and go up to - but not include - the input.

```
print("This will execute first")
for _ in range(3):
    print("This line will execute three times")
print("Now we are outside of the for loop!")
```

v/s

```
print("This will execute first")
for _ in [0,1,2]:
    print("This line will execute three times")
print("Now we are outside of the for loop!")
```

Accumulator Pattern

- One common programming "pattern" is to traverse a sequence, **accumulating** a value as we go, such as the sum-so-far or the maximum-so-far.
- That way, at the end of the traversal we have accumulated a single value, such as the sum total of all the items or the largest item.
- **The anatomy of the accumulation pattern includes:**
  - **initializing** an "accumulator" variable to an initial value (such as 0 if accumulating a sum)
  - **iterating** (e.g., traversing the items in a sequence)
  - **updating** the accumulator variable on each iteration (i.e., when processing each item in the sequence)

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
accum = 0
for w in nums:      [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    accum = accum + w
    print(accum)
```

NOTE :Remember that the key to making it work successfully is to be sure to initialize the variable before you start the iteration. Once inside the iteration, it is required that you update the accumulator.