01. - An Informational Introduction to Python In the following examples, input and output are distinguished by the presence or absence of prompts (>>> & ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command. You can use the "Copy" button (It appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the line into your interpreter. Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with a Hash character "#" and extend to the end of the physical line. A comment may appear at the start of a line or following whitespaces or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

In [2]:
```
#This is the First comment
spam = 1  #and this is the second comment
          # ... and now the Third!
text = "#This is not comment becouse it's inside the qoutes."
```

02- Using Python as a calculator Let's try some simple Python commands. Start the interpreter and wait for the primary prompts, >>>. (It should not take long) 03- Numbers The interpreter acts as a simple calculator: you can type an expression into it and it will write the value. Expression syntax is straightforward: the operators "+", "-", "*", and "/" can be used to perform arithmetic. Parantheses (()) can be used for grouping. For Example:r

In [3]:
```
>>> 2+2
```

Out[3]:
```
4
```

In [4]:
```
>>> 50-5*6
```

Out[4]:
```
20
```

In [5]:
```
>>> (50-5*6)/4
```

Out[5]:
```
5.0
```

In [6]:
```
>>> 8/5  #Division always returns a floating-point number
```

Out[6]:
```
1.6
```

The integer numbers (e.g., 2,4,20) have type int, whereas ones with a fractional part (e.g., 5.0, 1.6) have type float. We will see more about numeric types later in the tutorial. Division "/" always returns a float. To do floor division and get an integer result, you can use the "//" operator. To calculate the remainder, you can use "%"

In [7]:
```
>>> 17/3    #Classic division returns a float
```

Out[7]:
```
5.666666666666667
```

In [9]:
```
>>>
>>> 17//3   #floor division discards the fractional part
```

Out[9]:
```
5
```

In [11]:
```
>>> 17%3   #The % operator returns the remainder of the division
```

Out[11]:
```
2
```

In [12]:
```
>>> 5*3+2   # floored quotient * divisor + remainder
```

Out[12]:
```
17
```

With Python, it is possible to use the "**" operator to calculate powers

In [13]:
```
>>> 5**2   # 5 Sqoured
```

Out[13]: 25

In [14]:
```
>>> 2**7    # 2 to the power of 7
```

Out[14]: 128

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is diplayed before the next interactive prompt:

In [30]:
```
>>> width = 20
```

In [ ]:
```
>>> height = 5*9
```

In [29]:
```
>>> width * height
```

Out[29]: 900

If a variable is not "Defined" (assigned a value), trying to use it will give you an error

In [31]:
```
>>> #Try to access an undefined variable
```

Traceback (most call recent call last): File "<stdin>", line 1, in <module> NameError: name 'n' is not definedThere is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

In [35]:
```
>>> 4*3.75-1
```

Out[35]: 14.0

In interactive mode, the last printed expression is asssigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

In [36]:
```
>>> tax = 12.5/100
>>> price = 100.50
>>> price * tax
```

Out[36]: 12.5625

In [37]:
```
>>> price + _
```

Out[37]: 113.0625

In [38]:
```
>>> round(_, 2)
```

Out[38]: 113.06

This variable should be treated as read only by the user. Don't explicitly assign a value to it - you would create an independent local variable with the name masking the built-in varibale with its magic behavior. In adiition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g., 3+5j)Python can manipulate text(represented by the str, also called "strings") as well as numbers. This includes character "("!", words "rabbit", names "Paris", sentences "Got your back.". etc. "Yay!" :)". They can be enclosed in single quotes ('...') or double quotes ("...") with the same result.

In [39]:
```
>>> 'spam eggs' #Single Quotes
```

Out[39]: 'spam eggs'

In [40]:
```
>>> "Paris rabbit got your back :)! Yay!"  #Double Quotes
```

Out[40]: 'Paris rabbit got your back :)! Yay!'

In [44]:
```
>>> '1979'  #Digits and Numerals enclosed in quotes are also string
```

Out[44]:
```
'1979'
```

To Quote a Quote, we need to "Escape" it, by preceding it with \. Alternatively, we can use the other type of quotation marks:

In [45]:
```
>>> 'doesn\'t' #use \' to escape the signle qoute
```

Out[45]:
```
"doesn't"
```

In [46]:
```
>>> "doesn't"  #...or use double qoutes instead
```

Out[46]:
```
"doesn't"
```

In [47]:
```
>>> '"Yes," they saind.'
```

Out[47]:
```
'"Yes," they saind.'
```

In [48]:
```
>>> "\"Yes,\"They Said."
```

Out[48]:
```
'"Yes,"They Said.'
```

In [50]:
```
>>> '"Isn\'t,"They Said'
```

Out[50]:
```
'"Isn\'t,"They Said'
```

In the Python Shell, The String definition and output string can look different. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

In [52]:
```
>>> s = 'First line. \nSecond line.'  #\n means newline
>>> s   #Without print(), special character are included in the string
```

Out[52]:
```
'First line. \nSecond line.'
```

In [53]:
```
>>> print(s)   #With print(), special character are interpreted, so \n produces
```

```
First line.
Second line.
```

If you dont't want character prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote:

In [54]:
```
>>> print('C:\some\name')  #here \n means newline
```

```
C:\some
ame
```

In [55]:
```
>>> print(r'C:\some\name')   #note the r before the quote
```

```
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of \ character; see the FAQ entry for more information and workarounds.String litrals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''. End-of-line characters automatically included in the string, nut it's possible to prevent this by adding a \ at end of the line. In the following example, the initial newline is not included:

In [62]:
```
>>> print("""\
... Usage: Thingy [OPTIONS]
...        -h                                  Display the usage message
...        -H                                  Hostname to connect to
... """)
```

```
Usage: Thingy [OPTIONS]
       -h                                        Display the usage message
       -H                                        Hostname to connect to
```

Strings can be concatenated(Glued Together) with the + Operator, and repeated with *:

In [1]:
```
>>> # 3 Times 'un' followed by 'ium'
>>> 3 * 'um' + 'ium'
```

Out[1]:
```
'umumumium'
```

Two or more String literals (i.e., the ones enclosed between quotes) next to each other are automatically concatenated

In [2]:
```
>>> 'Py' 'thon'
```

Out[2]:
```
'Python'
```

This feature is particularly useful when you want to break long strings:

In [3]:
```
>>> text = ('Put Several strings within parentheses '
            'to have them joined together.')
>>> text
```

Out[3]:
```
'Put Several strings within parentheses to have them joined together.'
```

This only works with two literals, though, not with variables or expressions:

In [1]:
```
>>> prefix = 'Py'
>>> prefix = 'thon'   #can't concatenate a variable and a string literal
    File "<stdin>", line 1
        prefix 'thon'
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
    File "<stdin>", line 1
        ('un' * 3) 'ium'
SyntaxError: invalid syntax
```

```
  Cell In[1], line 3
    File "<stdin>", line 1
    ^
IndentationError: unexpected indent
```

If you want to concatenate variables or a variable and a literal, use +:

In [7]:
```
>>> prefix = 'Py'
>>> prefix + 'thon'
```

Out[7]:
```
'Python'
```

Strings can be indexed (subscripted), with the first character having index 0. There is no separate character type a character is simply a string of size one:

In [8]:
```
>>> word = 'Python'
>>> word[0]  #Character at position 0
```

Out[8]:
```
'P'
```

In [9]:
```
>>> word[5]  #Character at position 5
```

Out[9]:
```
'n'
```

Indexes may also be negative numbers, to start counting from the right:

In [12]:
```
>>> word[-1]  #Last Character
```

```
Out[12]: 'n'
```

```
In [13]: >>> word[-2]  #Second-Last Character
```

```
Out[13]: 'o'
```

```
In [14]: >>> word[-6]  #1st Character
```

```
Out[14]: 'P'
```

Note that since -0 is the same as 0, negative indexes start from -1In addition to indexing, slicing is also supported. While indexing is used to obtain individual characters, slicing allows you to obtain a substring:

```
In [17]: >>> word[0:2]  #Character from position 0 (included) to 2 (Excluded)
```

```
Out[17]: 'Py'
```

```
In [18]: >>> word[2:5]  #Character from position 2 (included) to 5 (Excluded)
```

```
Out[18]: 'tho'
```

Slice Indexes have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the being sliced.

```
In [19]: >>> word[:2] #Character from the beginning to Position 2 (Excluded)
```

```
Out[19]: 'Py'
```

```
In [20]: >>> word[4:] #Character from Position 4 (Included) to the End
```

```
Out[20]: 'on'
```

```
In [23]: >>> word[-2:] #Character from the Second-last (Included) to the End
```

```
Out[23]: 'on'
```

Note: The start is always included, and the end is always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

```
In [24]: >>> word[:2] + word[2:]
```

```
Out[24]: 'Python'
```

```
In [25]: >>> word[:4] + word[4:]
```

```
Out[25]: 'Python'
```

One Way to remember how slices work is to think of the indexes as pointing between characters, with the left edge of the first character numbered 0. Then the right edge of the last character of string of n characters has index n, for example: +---+---+---+---+---+---+ | P | y | t | h | o | n | +---+---+---+---+---+---+ 0 1 2 3 4 5 6 -6 -5 -4 -3 -2 -1The First row of Numbers gives the position of the indexes 0 to 6 in the string; the second row shows the corresponding negative indexes. The slice from i to j consistsm of all characters between the edge labeled i and j, respectivelyFor non-negative indexes, the length of a slice is the difference between the indexes, if both are within bounds. for example, the lenght word[1:3] is 2Attempting to use an index that is too large will result in an error:

```
In [31]: >>> word[42]  #The word only has 6 charcaters
         Traceback (most recent call last)
             File "<stdin>", line 1, in <module>
         IndexError: string index out of range
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[31], line 1
----> 1 word[42]

IndexError: string index out of range
```

Howeever, Out of Range Slice indexes are handled gracefully when used for slicing:

In [32]:
```
>>> word[4:42]
```

Out[32]:
```
'on'
```

In [34]:
```
>>> word[42:]
```

Out[34]:
```
''
```

Pyhton strings cannot be changed - they are immutable. Therefore, assigning to an indexed position in the string results in an error.

In [ ]:
```
>>> word[0] = 'J'
Traceback (most recent call last)
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last)
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one

In [4]:
```
>>> word = 'Python'
>>> 'J' + word[1:]
```

Out[4]:
```
'Jython'
```

In [6]:
```
>>> word[:2] + 'py'
```

Out[6]:
```
'Pypy'
```

The Built-in function len() returns the length of a string:

In [7]:
```
>>> s = 'AkashSenFromDistrictMahobaUttarPradeshIndia'
>>> len(s)
```

Out[7]:
```
43
```

See Also: Text Sequence Type - str Strings are examples of Sequence types, and support the common operations supported by such types. String Methods Strings support a large number of methods for basic transformation and searching. f-strings String literals that have embedded expressions. Format String Syntax Information about string formatting with str.format(). printf-style String formatting The old formatting operations invoked when strings are the left operand of the % operator are described in more details here.Python knows several compound data types, used to group other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. The list might contain items of different types, but usually the items all have the same type

In [10]:
```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
```

Out[10]:
```
[1, 4, 9, 16, 25]
```

Like strings (And All Other Built-in sequence types), lists can be indexe amd sliced

In [11]:
```
>>> squares[0]   #INdexing returns the items
```

Out[11]:
```
1
```

In [12]:
```
>>> squares[-1]
```

Out[12]:
```
25
```

In [13]:
```
>>> squares[-3:]   #Slicing returns the items
```

Out[13]:
```
[9, 16, 25]
```

Lists also support operations like concatenation:

In [15]:
```
>>> squares + [36, 49, 64, 81, 100]
```

Out[15]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike immutable strings, lists are a mutable type, i.e., it is possible to change their content:

In [16]:
```
>>> cubes = [1, 8, 27, 65, 100]  #Something's wrong here
>>> 4 ** 3 #the cube of 4 is 64, not 65!
```

Out[16]:
```
64
```

In [17]:
```
>>> cubes[3] = 64  #Replace the wrong value
```

In [18]:
```
>>> cubes
```

Out[18]:
```
[1, 8, 27, 64, 100]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the existing list. Any changes you make to the list through one variable will be seen through all other varibales that refer to it.:

In [19]:
```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba) #They reference the same object
```

Out[19]:
```
True
```

In [27]:
```
>>> rgba.append("Alph")
>>> rgb
```

Out[27]:
```
['Red',
 'Green',
 'Blue',
 'Alph',
 'Alph',
 'Alph',
 'Alph',
 'Alph',
 'Alph',
 'Alph',
 'Alph']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list

In [33]:
```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
```

Out[33]:
```
['Red', 'Green', 'Alpha']
```

In [32]:
```
>>> rgba
```

Out[32]:
```
['Red', 'Green', 'Blue']
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

In [4]:
```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
```

Out[4]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

In [5]:
```
>>> #Replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
```

Out[5]:
```
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

In [6]:
```
>>> #Now remove them
>>> letters[2:5] = []
>>> letters
```

Out[6]:
```
['a', 'b', 'f', 'g']
```

In [7]:
```
>>> #CLear the list by replacing all the elements with an empty List
>>> letters[:] = []
>>> letters
```

Out[7]:
```
[]
```

The built-in function len() also applies to lists:

In [9]:
```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> len(letters)
```

Out[9]:
```
7
```

It is possible to nest lists(Create lists containing other lists), for example:

In [12]:
```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
```

Out[12]:
```
[['a', 'b', 'c'], [1, 2, 3]]
```

In [13]:
```
>>> x[0]
```

Out[13]:
```
['a', 'b', 'c']
```

In [14]:
```
>>> x[0][1]
```

Out[14]:
```
'b'
```

First Steps Towards Programming Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci as follows:

In [15]:
```
>>> #Fibonacci series:
>>> #The SUm of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
```

```
    print(a)
    a, b = b, a + b
```

```
0
1
1
2
3
5
8
```

This example introduces several new features. The first line contains multiple assignments: the variables a and b simultaneously get the new values 0 and 1. On the last, this was used again. Demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right. The while loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value; in fact, any sequence; anything with a non-zero length is true, and an empty sequence is false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: '<' (Less than), '>' (greater than), '==' (equal to), '<=' (less than or equal to), '>=' (greater than or equal to), and '!=' (not equal to). The Body of the loop is indented: Indentation is Python's way of grouping statements. AI the interactive prompt, you have to type a tab or spaces(s) for each indented line. In practice, you will prepare more complicated input for Python with the text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount. The print() function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can things nicely, like this:

In [6]:
```
>>> i = 256 * 256
>>> print('The value of i is = ', i)
```

```
The value of i is =  65536
```

The Keyword argument end can be used to avoid the newline after the output, or end the output with a different string:

In [9]:
```
>>> a, b = 0, 1
>>> while a < 1000:
    print(a, end = ',')
    a, b = b, a + b
```

```
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

In [ ]:
```
Footnotes

1- Since ** has higher precedence than -, -3**2 will be interpreted as -(3**2) a

2- Unlike other languages, special characters such as \n have the same meaning w
```