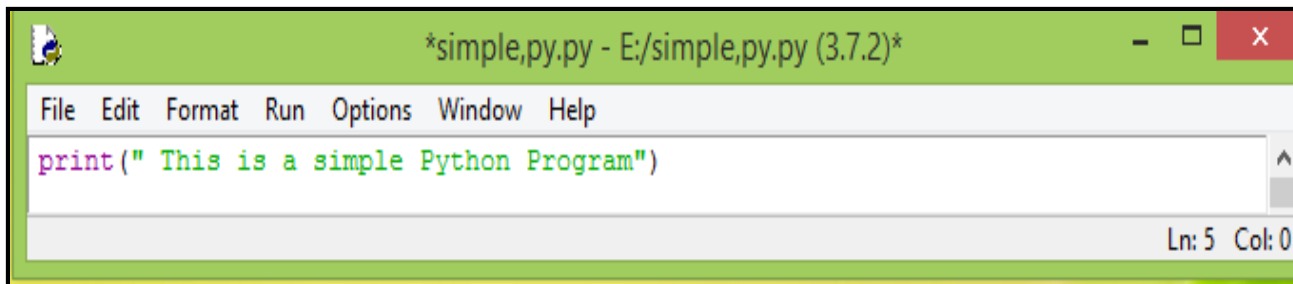


Machine Learning

The Basics : Writing a Python Program

Python programs must be written with a particular structure. The syntax must be correct, or the interpreter will generate error messages and not execute the program.



```
print("This is a simple Python Program")
```

We will consider two ways in which we can run the program (simple.py):

1. enter the program directly into IDLE's interactive shell and
2. enter the program into IDLE's editor, save it, and run it.

IDLE's interactive shell.

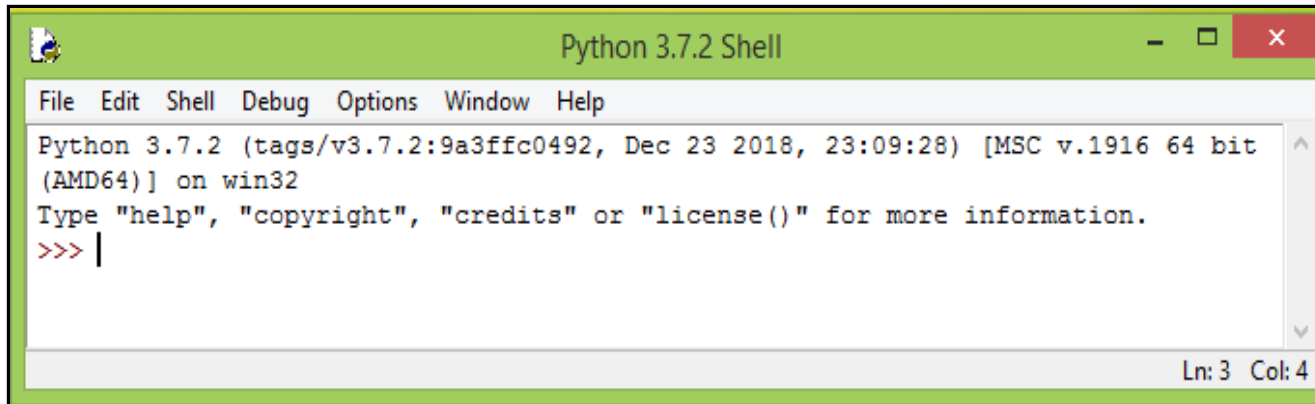
IDLE is a simple Python integrated development environment(**IDE**) available for Windows, Linux, and Mac OS X. You may type the previous one line Python program directly into IDLE and press enter to execute the program. The next figure shows the result using the IDLE interactive shell.

Since it does not provide a way to save the code you enter, the interactive shell is not the best tool for writing larger programs. The IDLE interactive shell is useful for experimenting with small snippets of Python code.

IDLE's editor. IDLE has a built in editor. From the IDLE menu, select New Window,. Type the program (simple.py) into the editor.

We can save your program using the Save option in the File menu. Save the code to a file named **simple.py**. The actual name of the file is irrelevant, but the name "simple" accurately describes the nature of this program. The extension **.py** is the extension used for Python source code. We can run the program from within the IDLE editor by pressing the **F5** function key or from the editor's Run menu: Run! **Run Module**. The output appears in the IDLE interactive shell window.

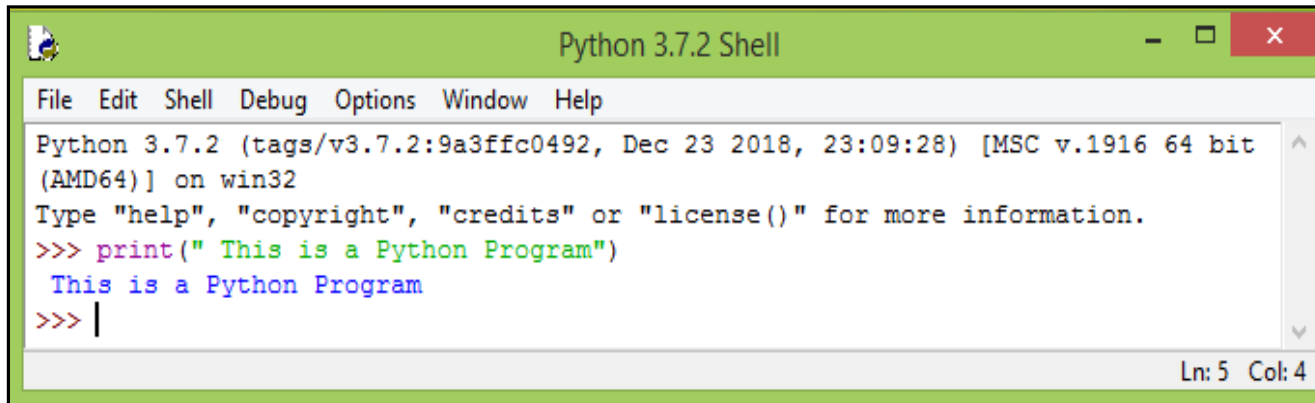
Introduction to Python Programming



```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Ln: 3 Col: 4

The IDLE interpreter Window.

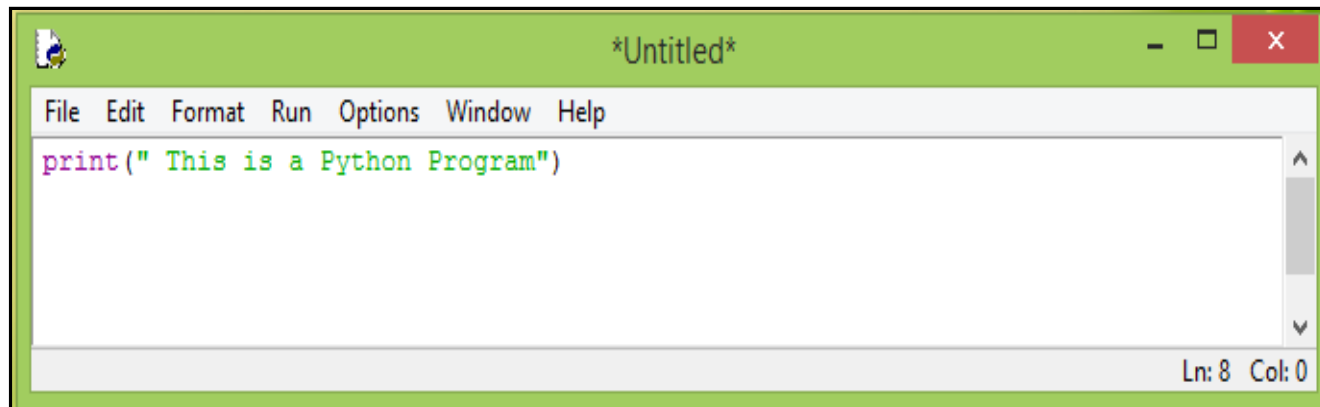


```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print(" This is a Python Program")
This is a Python Program
>>> |
```

Ln: 5 Col: 4

A simple Python program entered and run with the IDLE interactive shell

The editor allows us to save our programs and conveniently make changes to them later. The editor understands the syntax of the Python language and uses different colors to highlight the various components that comprise a program. Much of the work of program development occurs in the editor.



A simple Python program typed into the IDLE editor

A Longer Python program

More interesting programs contain multiple statements. In the next program (arrow.py), six print statements draw an arrow on the screen:

```
print(" * ")
print(" *** ")
print(" ***** ")
print(" * ")
print(" * ")
print(" * ")
```

The output of arrow.py will be :

```
*
***
*****
*
*
*
```

Values and Variables

Now we explore some building blocks that are used to develop Python programs.

We experiment with the following concepts:

- numeric values
- variables
- assignment
- identifiers
- reserved words

Integer Values

The number four (**4**) is an example of a numeric value. In mathematics, 4 is an integer value. Integers are whole numbers, which means they have no fractional parts, and they can be positive, negative, or zero.

Examples of integers include 4, -19, 0, and -1005. In contrast, 4.5 is not an integer, since it is not a whole number.

Python supports a number of numeric and non-numeric values. In particular, Python programs can use integer values. The Python statement

```
print(4)
```

prints the value **4**.

The number 4 by itself is not a complete Python statement and, therefore, cannot be a program. The interpreter, however, can evaluate a Python expression. You may type the enter 4 directly into the interactive interpreter shell:

```
>>> 4  
4  
>>>
```


Values and Variables

Python uses the **+** symbol with integers to perform normal arithmetic addition, so the interactive shell can serve as a handy adding machine:

```
>>> 3 + 4
7
>>> 1+2+4+10+3
20
>>> print (1+2+4+10+3)
20
```

The last line evaluated shows how we can use the **+** symbol to add values within a print statement that could be part of a Python program.

Consider what happens if we use quote marks around an integer:

```
>>> 99
99
>>> "99"
'99'
>>> '99'
'99'
```

Values and Variables

Notice how the output of the interpreter is different. The expression "99" is an example of a string value. A string is a sequence of characters. Strings most often contain non-numeric characters:

```
>>> "Ronaldo"  
'Ronaldo'  
>>> 'Ronaldo'  
'Ronaldo'
```

Python recognizes both single quotes (') and double quotes (") as valid ways to delimit a string value.

It is important to note that the expressions **44** and **'44'** are different.

One is an **integer** expression and the other is a **string** expression. All expressions in Python have a type. The type of an expression indicates the kind of expression it is. An expression's type is sometimes denoted as its class.

Values and Variables

The built in type function reveals the type of any Python expression:

```
>>> type(44)
<class 'int'>
>>> type('44')
<class 'str'>
```

Python associates the type name **int** with **integer** expressions and **str** with **string** expressions. The built in int function converts the string representation of an integer to an actual integer, and the str function converts an integer expression to a string:

```
>>> 44
44
>>> str(44)
'44'
>>> '55'
'55'
>>> int('55')    The expression str(44) evaluates to the string value '44', and
55               int('55') evaluates to the integer value 55.
```

Values and Variables

The plus operator (+) works differently for strings;

```
>>> 55 + 20
75
>>> '55' + '20'
'5520'
>>> 'Leonel' + 'Messi'
'LeonelMessi'
```

As you can see, the result of the expression `55 + 20` is very different from `'55' + '20'`. The plus operator splices two strings together in a process known as **concatenation**. Mixing the two types directly is not allowed:

```
>>> 55 + int('20')
75
>>> '55' + str(20)
'5520'
```

Values and Variables

The type function can determine the type of the most complicated expressions:

```
>>> type(44)
<class 'int'>
>>> type('44')
<class 'str'>
>>> type(44 + 77)
<class 'int'>
>>> type('44' + '77')
<class 'str'>
>>> type(int('37' + int(73)))
Class 'int'>
```

Commas may not appear in Python integer values. The number one thousand, five hundred twenty-eight would be written 1528, not 1,528.

Variables and Assignment

In algebra, variables represent numbers. The same is true in Python, except Python variables also can represent values **other** than numbers. The next program (variable.py) uses a variable to store an integer value and then prints the value of the variable.

```
x = 40  
print(x)
```

x = 40

This is an assignment statement. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol **=** which is known as the assignment operator. The statement assigns the integer value 40 to the variable x.

Variables and Assignment

```
print(x)
```

This statement prints the variable x's current value. Note that the lack of quotation marks here is very important. If x has the value 40, the statement

```
print(x)
```

prints 40, the value of the variable x, but the statement

```
print('x')
```

prints x, the message containing the single letter x.

Variables and Assignment

Variables can be reassigned different values as needed, as the next program(multipleassignment.py) shows.

```
x = 20
print('x = ' + str(x))
x = 40
print('x = ' + str(x))
x = 60
print('x = ' + str(x))
```

Observe that each print statement in the program (multipleassignment.py) is identical, but when the program runs (as a program, not in the interactive shell) the print statements produce different results:

```
x=20
x=40
x=60
```


Variables and Assignment

A programmer may assign multiple variables in one statement using **tuple** assignment. The next program (tupleassign.py) shows how:

```
x = 100
x, y, z = 1000, -475, 200
print('x =', x, ' y =', y, ' z =', z)
```

The output.

```
x = 1000 y = -475 z = 200
```

A tuple is a comma separated list of expressions. In the assignment statement

```
x, y, z = 1000, -475, 200
```

x, y, z is one tuple, and 1000, -475, 200 is another tuple.

Variables and Assignment

Python has strict rules for variable names. A variable name is one example of an identifier. An identifier is a word used to name things. One of the things an identifier can name is a variable.

and	del	from	None	try
as	elif	global	nonlocal	True
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Python keywords

Variables and Assignment

The most important thing to remember about variables names is that they should be well chosen. A variable's name should reflect the variable's purpose within the program. For example, consider a program controlling a point-of-sale terminal. The variable keeping track of the total cost of goods purchased might be named **total** or **total_cost**.

Variables and Assignment

Floating-point Types

Many computational tasks require numbers that have fractional parts. For example, to compute the area of a circle given the circle's radius, the value **pi** or approximately **3.14159** is used.

Python supports such non-integer numbers, and they are called **floating point** numbers. The name implies that during mathematical calculations the decimal point can move or "**float**" to various positions within the number to maintain the proper number of significant digits. The Python name for the floating-point type is **float**.

Consider the following interactive session:

```
>>> x= 7.72
>>> x
7.72
>>> type(x)
<class 'float'>
```

Variables and Assignment

Floating-point Types

The range of floating-points values (smallest value to largest value, both positive and negative) and precision (the number of digits available) depends of the Python implementation for a particular machine.

The next program (pi-print.py) prints an approximation of the mathematical value **pi**.

```
pi = 3.14159;  
print("Pi =", pi)  
print("or", 3.14, "for short")
```

The first line in (pi-print.py) prints the value of the variable pi, and the second line prints a literal value. Any literal numeric value with a decimal point in a Python program automatically has the type float.

Control Codes within Strings

The characters that can appear within strings include letters of the alphabet (A-Z, a-z), digits (0-9), punctuation (., :, ,, etc.), and other printable symbols (#, &, %, etc.). In addition to these “normal” characters, we may embed special characters known as control codes.

Control codes control the way text is rendered in a console window or paper printer. The backslash symbol (\) signifies that the character that follows it is a control code, not a literal character.

The string `'\n'` thus contains a single control code. The backslash is known as the escape symbol, and in this case we say the `n` symbol is escaped. The `\n` control code represents the newline control code which moves the text cursor down to the next line in the console window.

Other control codes include `\t` for tab, `\f` for a form feed (or page eject) on a printer, `\b` for backspace, and `\a` for alert (or bell). The `\b` and `\a` do not produce the desired results in the IDLE interactive shell, but they work properly in a command shell.

Control Codes within Strings

The next program (specialchars.py) prints some strings containing some of these control codes.

```
print('A\nB\nC')  
print('D\tE\tF')  
print('WX\bYZ')
```

When executed in a command shell, specialchars.py produces

```
A  
B  
C  
D      E      F  
WXZ
```

A string with a single quotation mark at the beginning must be terminated with a **single quote**; similarly, A string with a double quotation mark at the beginning must be terminated with a **double quote**. A single-quote string may have embedded double quotes, and a double-quote string may have embedded single quotes.

Control Codes within Strings

The next program(escapequotes.py) shows the various ways in which quotation marks may be embedded within string literals.

```
print("Did you know that 'Sample' is a word?")  
print('Did you know that "Sample " is a word?')  
print('Did you know that \' Sample \' is a word?')  
print("Did you know that \" Sample \" is a word?")
```

The output is :

```
Did you know that 'Sample' is a word?  
Did you know that "Sample " is a word?  
Did you know that ' Sample ' is a word?  
Did you know that " Sample" is a word?
```

Since the backslash serves as the escape symbol, in order to embed a literal backslash within a string you must use two backslashes in succession.

User Input

The print function enables a Python program to display textual information to the user.

Programs may use the input function to obtain information from the user. The simplest use of the input function assigns a string to a variable:

```
x = input()
```

The parentheses are empty because, the input function does not require any information to do its job.

The next program(usinginput.py) demonstrates that the input function produces a string value.

User Input

```
print('Enter some text:')  
x = input()  
print('Text entered:', x)  
print('Type:', type(x))
```

The following shows a sample run of the program (using `input.py`):

```
Enter some text:  
My name is Ronaldo  
Text entered: My name is Ronaldo  
'Type: <class 'str'>
```

The second line shown in the output is entered by the user, and the program prints the first, third, and fourth lines. After the program prints the message 'Please enter some text:', the program's execution stops and waits for the user to type some text using the keyboard. The user can type, backspace to make changes, and type some more. The text the user types is not committed until the user presses the enter (or return) key.

User Input

Sometimes, we want to perform calculations and need to get numbers from the user. The input function produces only strings, but we can use the **int** function to convert a properly formed string of digits into an integer.

The next program (addintegers.py) shows how to obtain an integer from the user.

```
print('Please enter an integer value:')  
x = input()  
print('Please enter second integer value:')  
y = input()  
num1 = int(x)  
num2 = int(y)  
print(num1, '+', num2, '=', num1 + num2)
```

The output :

```
Please enter an integer value:  
24  
Please enter second integer value:  
36  
24 + 36 =60
```

User Input

The previous program (addintegers.py) can be expressed more compactly using this form of the input function as shown in the next program (addintegers2.py).

```
x = input('Please enter an integer value: ')
y = input('Please enter another integer value: ')
num1 = int(x)
num2 = int(y)
print(num1, '+', num2, '=', num1 + num2)
```

User Input

The next program(addintegers3.py) is even shorter than the previous programs. It combines the **input** and **int** functions into one statement.

```
num1 = int(input('Please enter an integer value: '))  
num2 = int(input('Please enter another integer value: '))  
print(num1, '+', num2, '=', num1 + num2)
```

int(input('Please enter an integer value: '))

uses a technique known as **functional composition**. The result of the input function is passed directly to the **int function** instead of using the intermediate variables like the previous programs.

The eval Function

The input function produces a string from the user's keyboard input. If we wish to treat that input as a number, we can use the int or float function to make the necessary conversion:

```
x = float(input('Please enter a number'))
```

Here, whether the user enters 20 or 20.0, x will be a variable with type floating point. What if we wish x to be of type integer if the user enters 20 and x to be floating point if the user enters 20.0?

Python provides the eval function that attempts to evaluate a string in the same way that the interactive shell would evaluate it.

The next function (evalfunc.py) illustrates the use of **eval**.

The eval Function

```
x1 = eval(input('Entry x1? '))  
print('x1 =', x1, ' type:', type(x1))
```

```
x2 = eval(input('Entry x2? '))  
print('x2 =', x2, ' type:', type(x2))
```

```
x3 = eval(input('Entry x3? '))  
print('x3 =', x3, ' type:', type(x3))
```

```
x4 = eval(input('Entry x4? '))  
print('x4 =', x4, ' type:', type(x4))
```

```
x5 = eval(input('Entry x5? '))  
print('x5 =', x5, ' type:', type(x5))
```

The eval Function

A sample run of the program (evalfunc.py) produces

```
Entry x1? 10
x1 = 10  type: <class 'int'>
Entry x2? 20
x2 = 20  type: <class 'int'>
Entry x3? 'x1'
x3 = x1  type: <class 'str'>
Entry x4? x1
x4 = 10  type: <class 'int'>
Entry x5? x6
Traceback (most recent call last):
  File "E:/Programs/checking.py", line 13, in <module>
    x5 = eval(input('Entry x5? '))
  File "<string>", line 1, in <module>
NameError: name 'x6' is not defined
```


The eval Function

Notice that when the user enters **4**, the variable's type is **integer**.

When the user enters **4.0**, the variable is a **floating-point** variable.

For x3, the user supplies the string 'x3' (note the quotes), and the variable's type is **string**.

The more interesting situation is x4. The user enters x1 (no quotes). The **eval** function evaluates the non-quoted text as a reference to the name x1. The program bound the name x1 to the value 4 when executing the first line of the program.

Finally, the user enters x6 (no quotes). Since the quotes are missing, the eval function does not interpret x6 as a literal string; instead eval treats x6 as a name and attempts to evaluate it. Since no variable named x6 exists, the eval function prints an error message.

The eval Function

The eval function dynamically translates the text provided by the user into an executable form that the program can process. This allows users to provide input in a variety of flexible ways; for example, users can enter multiple entries separated by commas, and the eval function evaluates it as a Python tuple. As the next program (addintegers4.py) shows, this makes tuple assignment possible.

```
num1, num2 = eval(input('Please enter number 1, number 2: '))  
print(num1, '+', num2, '=', num1 + num2)
```

The following sample run shows how the user now must enter the two numbers at the same time separated by a comma:

```
'Please enter number 1, number 2: 10, 20  
10 + 20 =30
```

Controlling the print Function

In the program(`addintegers.py`) we would prefer that the cursor remain at the end of the printed line so when the user types a value it appears on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor automatically will move down to the next line.

The print function as we have seen so far always prints a line of text, and then the cursor moves down to the next line so any future printing appears on the next line. The print statement accepts an additional argument that allows the cursor to remain on the same line as the printed text:

```
print('Please enter an integer value:', end='')
```

The expression **`end=""`** is known as a **keyword** argument. The term keyword here means something different from the term keyword used to mean a reserved word.

Controlling the print Function

Another way to achieve the same result is :

```
print(end='Please enter an integer value: ')
```

This statement means "Print nothing, and then terminate the line with the string 'Please enter an integer value:' rather than the normal `\n` newline code. The behaviour of the two statements is almost same.

```
print()
```

essentially moves the cursor down to next line.

Controlling the print Function

By default, the print function places a single space in between the items it prints.

Print uses a keyword argument named **sep** to specify the **string** to use insert between items. The name **sep** stands for **separator**. The default value of sep is the string ' ', a string containing a single space.

The next program (printsep.py) shows the sep keyword customizes print's behavior.

```
w, x, y, z = 10, 15, 20, 25
print(w, x, y, z)
print(w, x, y, z, sep=',')
print(w, x, y, z, sep='')
print(w, x, y, z, sep=':')
print(w, x, y, z, sep='-----')
```

The output of the program is :

```
10 15 20 25
10,15,20,25
10152025
10:15:20:25
10-----15-----20-----25
```

Controlling the print Function

```
10 15 20 25  
10,15,20,25  
10152025  
10:15:20:25  
10-----15-----20-----25
```

The first of the output shows print's default method of using a **single space** between printed items.

The second output line uses **commas** as separators.

The third line runs the items together with an **empty string** separator.

The fourth line have **:** as the separator.

The fifth line shows that the separating string may consist of **multiple characters**.

Thank You