Introduction to Engineering Technology Programming Concepts

Think about some of the different ways that people use computers. The uses of computers are almost limitless in our everyday lives. **Computers can perform such a wide variety of tasks because they can be programmed.** This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. **A *program* is a set of instructions that a computer follows to perform a task.**

Programs are commonly referred to as *software*. Software is essential to a computer because it controls everything the computer does. All of the software that we use to make our computers useful is created by individuals working as programmers or software developers.

A *programmer,* or *software developer,* is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers' work used in business, medicine, government, law enforcement, agriculture, academics, entertainment, and many other fields.

Learning basic computer **programming** skills is **important** for **engineering** and **technology students** in their early years of college education. It is believed that with the modular **programming** strategy and the core **programming** skills, the **students** will be able to develop computer code to solve most **engineering** problems.

At Queensborough Community College, the department of Engineering offers Dual/Joint Degree Programs of ***Computer Science and Information Security A.S. degree***. This Dual/Joint Degree prepares students from programs that rely on computing and quantitative methods, especially in areas related to digital forensics and cybersecurity. All the 64 credits of Computer Science and Information Security A.S. degree is transferable to baccalaureate degree programs at John Jay College of Criminal Justice.

High-Level and Low-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a **low-level language**.

A **high-level language** allows you to create powerful and complex programs without knowing how the CPU works and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), he or she would write the following instruction to display the message *Hello world* on the computer screen: DISPLAY "Hello world"

Python is a modern, high-level programming language that we will use in this book. In Python you would display the message *Hello world* with the following instruction:

```
print('Hello world')
```

Compilers and Interpreters

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Depending on the language in which a program has been written, the programmer will use either a compiler or an interpreter to make the translation.

A *compiler* is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in **Figure 1-1**. As shown in the figure, compiling and executing are two different processes.

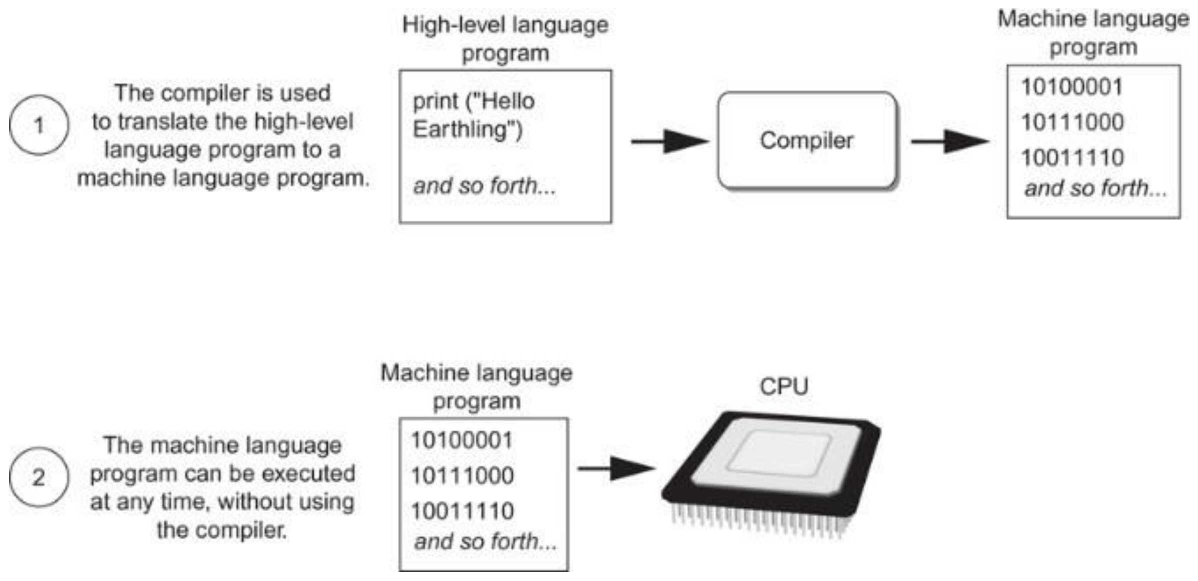

1
The compiler is used to translate the high-level language program to a machine language program.
High-level language program
print ("Hello Earthling")
and so forth...
Compiler
Machine language program
10100001
10111000
10011110
and so forth...
2
The machine language program can be executed at any time, without using the compiler.
Machine language program
10100001
10111000
10011110
and so forth...
CPU

Figure 1-1 Compiling Process

The Python language uses an ***interpreter*, which is a program that both translates and executes** the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to machine language instructions then immediately executes them. This process repeats for every instruction in the program. This process is illustrated in **Figure 1-2**. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

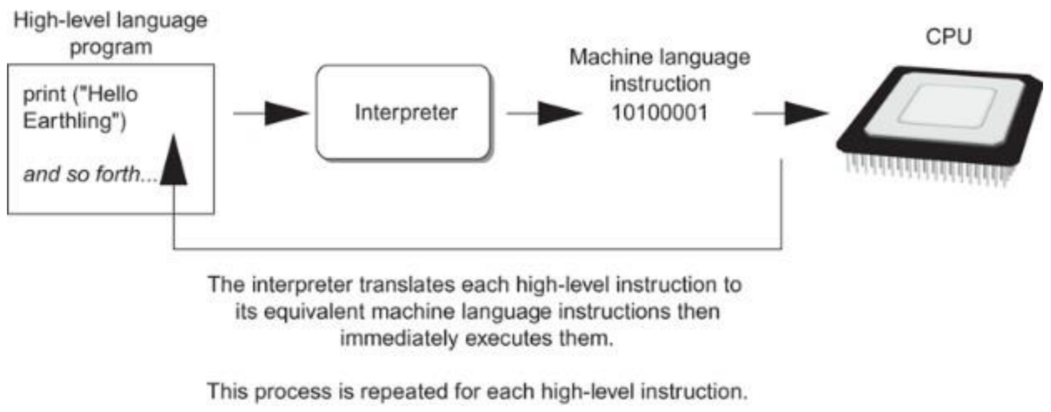

High-level language program
print ("Hello Earthling")
and so forth...
Interpreter
Machine language instruction
10100001
CPU
The interpreter translates each high-level instruction to its equivalent machine language instructions then immediately executes them.
This process is repeated for each high-level instruction.

Figure 1-2 Executing a high-level program with an interpreter

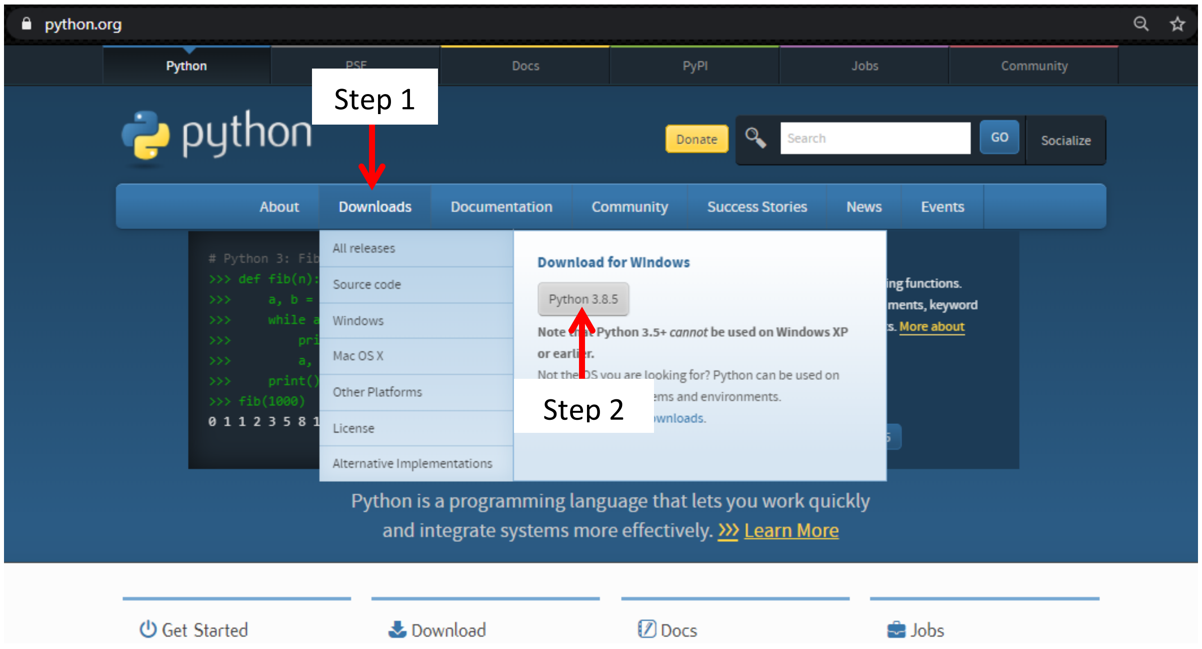
Introduction to python

A computer program is a set of instructions that causes a computer to perform some kind of action. Computers use multiple languages to communicate known as programming languages. A programming language is simply a particular way to talk to a computer in a set of instructions that both humans and the computer can understand. **Python** is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python became popular for the simplicity of its syntax and how fast its compiling process is. It offers concise and readable code by humans, which makes it easier to build models for machine learning. Not surprisingly, given its accessible and versatile nature, Python is among the top five most popular languages in the world. Python is used by Wikipedia, Google (where Van Rossum used to work), Yahoo!, CERN and NASA, among many other organizations. It is often used as a "scripting language" for web applications.

Installing python

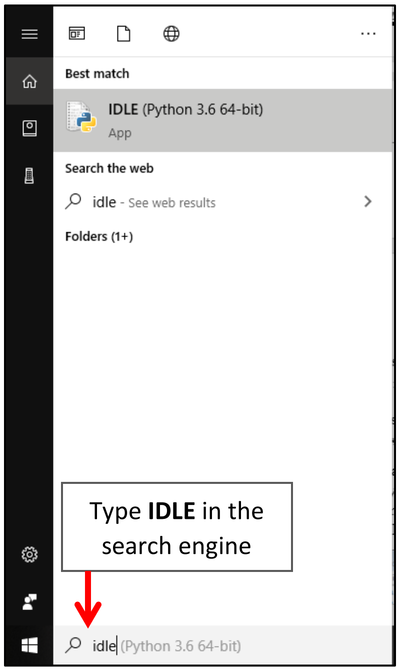
Installing python is fairly straightforward. To install python for Microsoft Windows or Mac, you can go to https://www.python.org/



Note that Python 3.5+ *cannot* be used on Windows XP or earlier.

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Download the python for Windows or Mac and follow the installation steps. Once the installation is completed, an IDLE python icon should appear in the computer search:

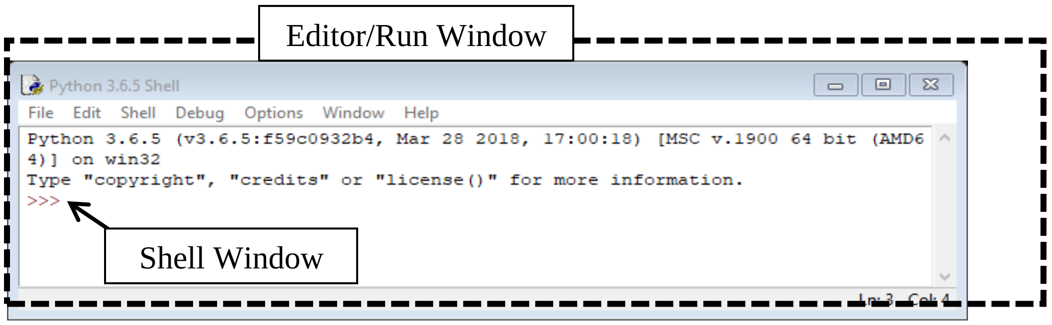


IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and Mac OS X
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

When double click on the icon, it should open the IDLE Python shell:



```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

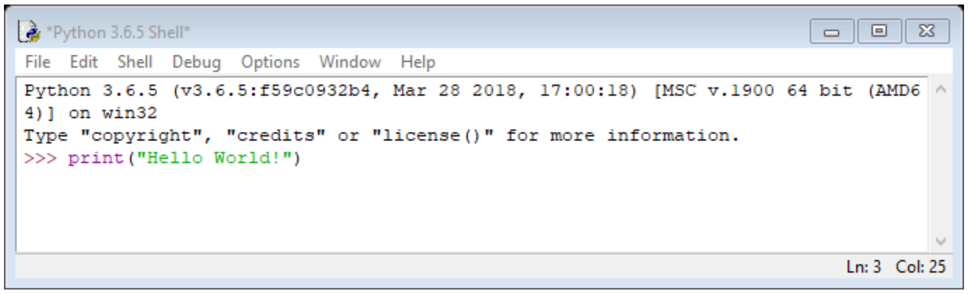
IDLE Python Shell

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. Output windows, such as used for Edit / Find in Files, are a subtype of edit window. They currently have the same top menu as Editor Windows but a different default title and context menu.

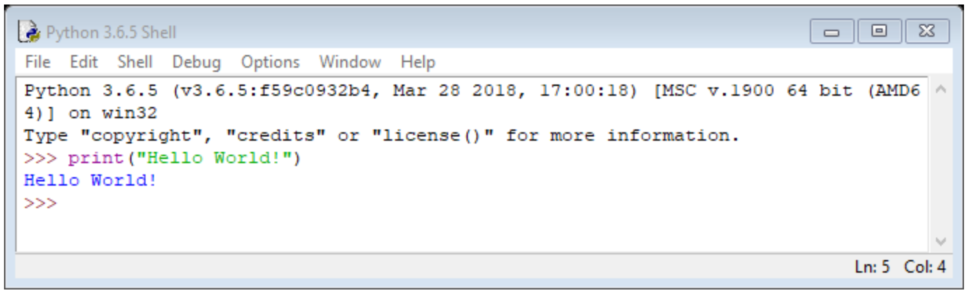
IDLE's menus dynamically change based on which window is currently selected. Each menu documented below indicates which window type it is associated with.

In the shell window, the three greater-than signs **>>>** are called the *prompt*. A command **prompt** is used in a text-based or "command-line" interface.

For example, the first command that we are learning in python is *print*. The *print* command is used to display out whatever is inside the parenthesis. If double quotation marks is used within the parenthesis, it means that the code is printing the strings in between the double quotation marks. Let's type the code: print("Hello World!")



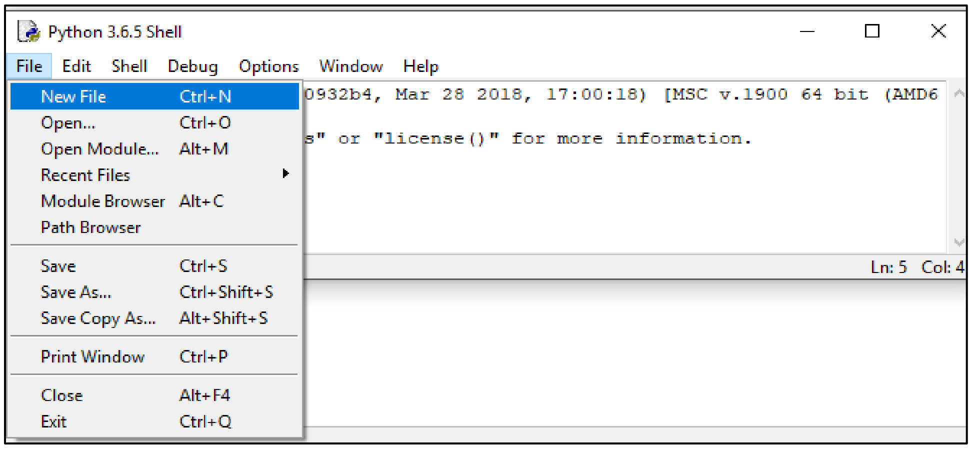
```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
```



```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

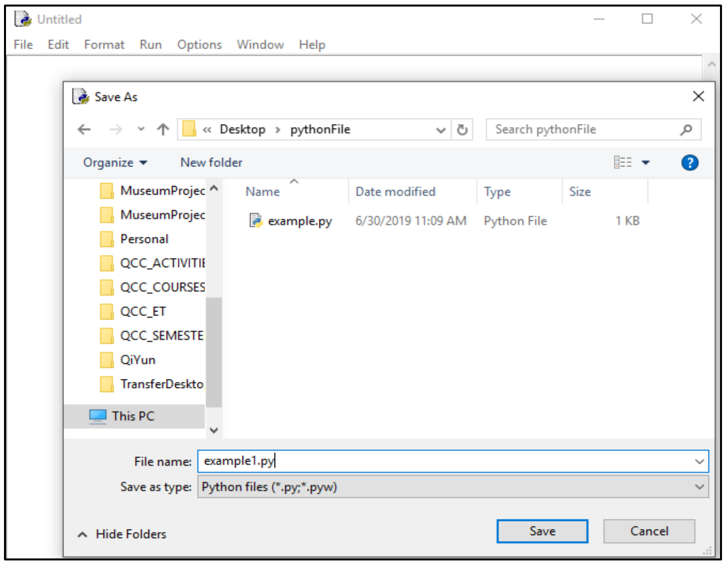
Saving a python program

Python file is saved with the file extension ***py***. To create a python file, select a *New File* from the File menu:



```
0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD6
s" or "license()" for more information.
```

Once the python file appears, we can save it first in a local folder and give a file name. During the saving process, make sure that the *Save As type* is selected as: ***Python file (.py)***

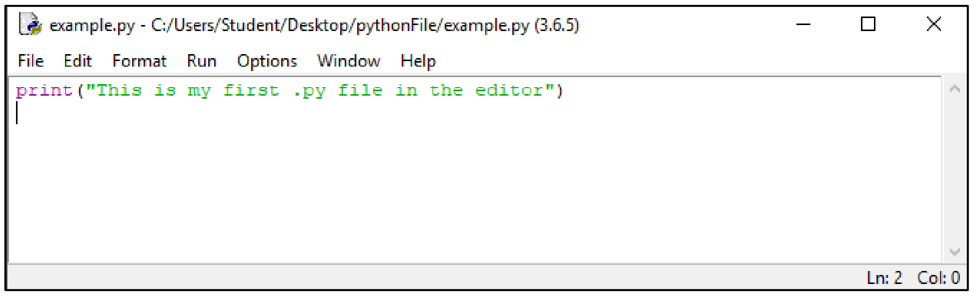


Name	Date modified	Type	Size
example.py	6/30/2019 11:09 AM	Python File	1 KB

File name: example1.py

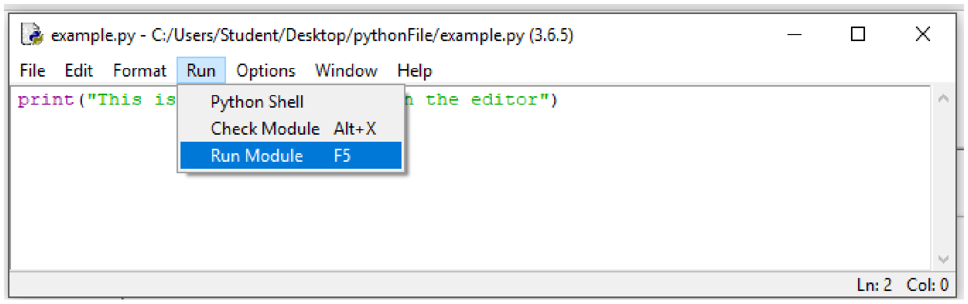
Save as type: Python files (*.py;*.pyw)

In the python file, we can write the code within the file:

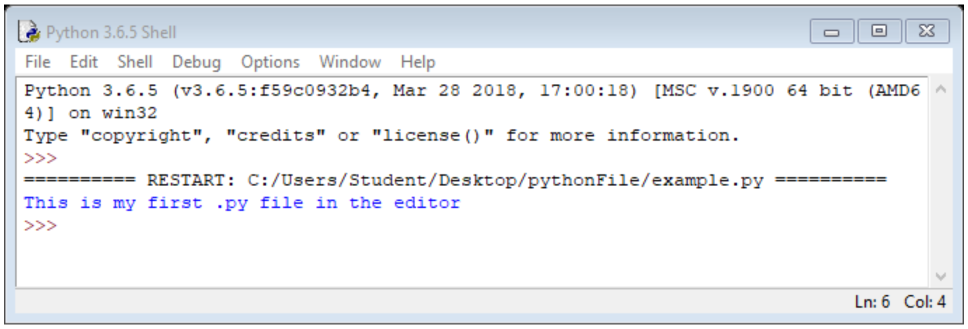

example.py - C:/Users/Student/Desktop/pythonFile/example.py (3.6.5)
File
Edit
Format
Run
Options
Window
Help
Ln: 2 Col: 0

```python
print("This is my first .py file in the editor")
```

To run the file, we save the file, and click on the Run tab and select Run Module, or use the F5 function key from the keyboard.


example.py - C:/Users/Student/Desktop/pythonFile/example.py (3.6.5)
File Edit Format Run Options Window Help
Python Shell
Check Module Alt+X
Run Module F5
Ln: 2 Col: 0

```
print("This is                n the editor")
```



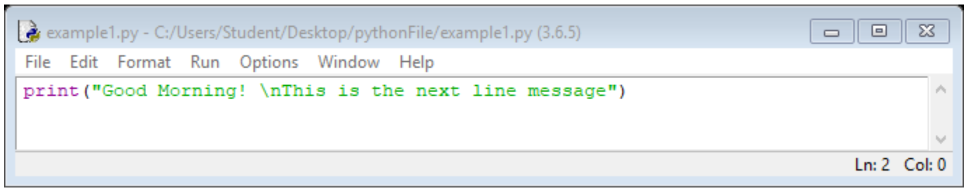
```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/Student/Desktop/pythonFile/example.py ==========
This is my first .py file in the editor
>>>
```

In Python strings, the backslash "\"

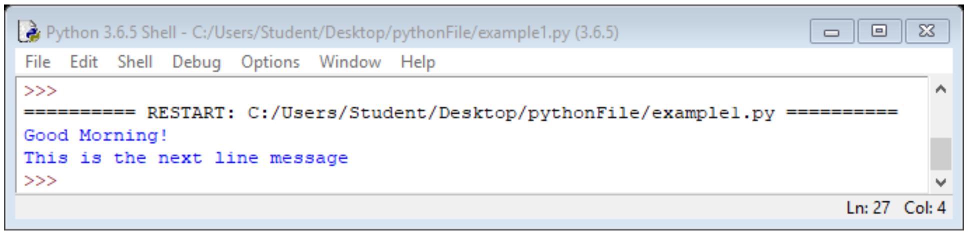
The backslash "\" is a **special character**, also called the "**escape**" character. It is used in representing certain whitespace characters:

- **\t** is a tab
- **\n** is a newline

For example, if we want to make a new line in a message, we can use the command **\n**


example1.py - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 0

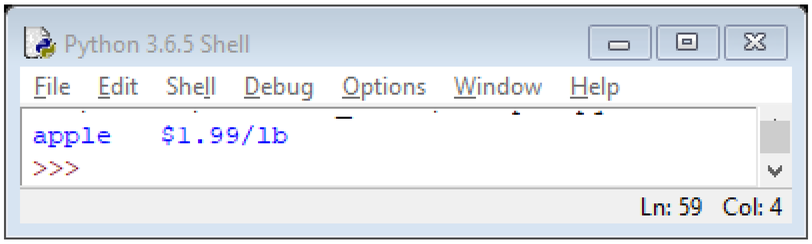
```
print("Good Morning! \nThis is the next line message")
```


Python 3.6.5 Shell - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Shell Debug Options Window Help
Ln: 27 Col: 4

```
>>>
========== RESTART: C:/Users/Student/Desktop/pythonFile/example1.py ==========
Good Morning!
This is the next line message
>>>
```

You can also add a tab to a line as:

```
print('apple\t$1.99/lb');
```

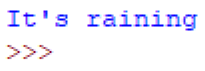


```
apple	$1.99/lb
>>>
```

Conversely, prefixing a special character with \ turns it into an ordinary character. This is called "escaping". For example, \ is the single quote character. `'It\'s raining'` therefore is a valid string and equivalent to `It's raining`.

```
    print ("It\ 's raining")
```

Prompt result:

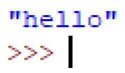


```
It's raining
>>>
```

Likewise, ' " ' can be escaped: `"\"hello\""` is a string begins and ends with the literal double quote character.

```
     print ("\"hello\"")
```

Prompt result:

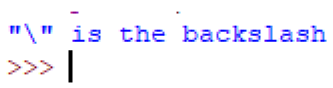


```
"hello"
>>>
```

Finally, "\" can be used to escape itself: "\\" is the literal backslash character.

```
     print ('"\\" is the backslash')
```

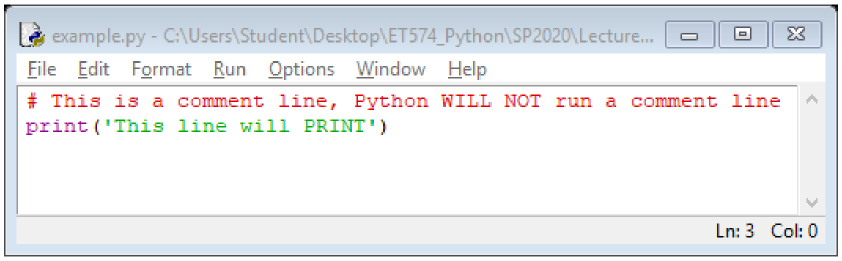
Prompt result:



```
"\" is the backslash
>>>
```

Comments

Comments in Python start with the hash character, # , and extend to the end of the physical line. A **comment** may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character.


example.py - C:\Users\Student\Desktop\ET574_Python\SP2020\Lecture...
File Edit Format Run Options Window Help
Ln: 3 Col: 0

```
# This is a comment line, Python WILL NOT run a comment line
print('This line will PRINT')
```

Prompt result

```
This line will PRINT
>>> |
```

You can add a multiple comment lines by using triple-quotes from the beginning to the end of the comment.

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Prompt result

```
Hello, World!
>>> |
```

Values and types

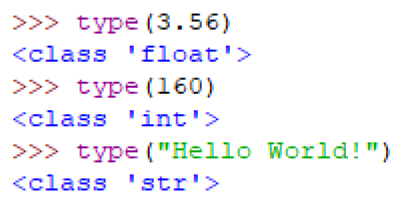
A value is one of the fundamental things—like a letter or a number—that a program manipulates. The values we have seen so far are 2(the result when we added 1 + 1), and ’Hello, World!’

These values belong to different types: 2 is an integer, and ’Hello, World!’ is a string, so-called because it contains a “string” of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

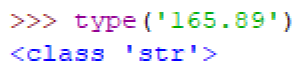
If you are not sure what type a value has, the interpreter can tell you.

Not surprisingly, strings belong to the type **`str`** and integers belong to the type **`int.`** Less obviously, numbers with a decimal point belong to a type called **`float`**, because these numbers are represented in a format called floating-point.



```
>>> type(3.56)
<class 'float'>
>>> type(160)
<class 'int'>
>>> type("Hello World!")
<class 'str'>
```

What about values like **'165.89'**? It look like an integer, but it is in between quotation marks like strings. Therefore 165.89 is a string.

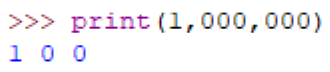


```
>>> type('165.89')
<class 'str'>
```

You can also print the value type as:

```
print(type(1.5))
```

When you type a large integer, you might be tempted to use commas between groups of three digits, as in **1,000,000**. This is not a legal integer in Python, but it is a legal expression:



```
>>> print(1,000,000)
1 0 0
```

Well, that's not what we expected at all! Python interprets **1,000,000** as a comma-separated list of three integers, which it prints consecutively. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

Calculations and Variables

Now that you have Python installed and familiar with the IDLE Python shell, we are ready to do something with Python. We will start with simple mathematical operations and then move on to variables.

Variables

Variable in programming describes a place to store information such as numbers, text, lists of numbers and text, and so on. Those information can be retrieved later for use by calling the variable's name.

For example, to create a variable named *number*, we use an equal sign (=) and then the type of information the variable should be the label for. If we want to assign number to 120 to variable *number*, we can write the code as: number = 120

Variable names

Variable names can be made up of letters, numbers, and the underscore character _, but they:

- **CAN'T** start with a number
- **CAN'T** have symbols, except underscore character

- **CAN'T** have space.
- **CAN'T** use Python keywords

  It turns out that `class` is one of the Python keywords. Keywords define the language's rules and structure, and they cannot be used as variable names.

  Python has twenty-nine keywords:

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

You might want to keep this list handy. If the interpreter displays an error about one of your variable names and you don't know why, see if it is on this list.

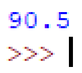
We can name our variable as we wish, but my recommendation is to name the variables according to its value or use.

Prompting a variable value

To prompt a variable value, we can just type the variable name:

```
x = 90.5
print (x)
```

Prompt result



```
90.5
>>> |
```

Using variables

Variables can also use amount them. For example, if we want to add to variables *number1* and *number2*, we can write the following code:

```
number1 = 20
number2 = 30
addition = number1 + number2
print ("The sum is: ", addition)
```

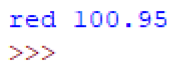
Also remember that variables are case sensible, for example, from the previous example, if we named the variable as *number1* and then we call the variable as *Number1,* we will have an error when we try to run the Python file, because *number1* and *Number1* is not the same variable.

Assign value to multiple variables

- Python allows us to assign values to multiple variables in one line. Each variable and values must be separated by a comma:

```
item1, item2, item3 = "red", 28, 100.95;
print(item1, item3);
```

Prompt result

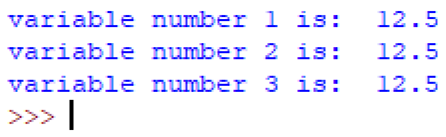


```
red 100.95
>>>
```

- You can assign the same value to multiple variables in one line:

```
number1=number2=number3= 12.5
print("variable number 1 is: ", number1)
print("variable number 2 is: ", number2)
print("variable number 3 is: ", number3)
```

Prompt result



```
variable number 1 is:  12.5
variable number 2 is:  12.5
variable number 3 is:  12.5
>>> 
```

input( ) function

Python user **input** from the keyboard can be read using the **`input()`** built-in function. The **`input`**( ) from the user is read as a string and can be assigned to a variable. After entering the value from the keyboard, we have to press the "Enter" button. Then the **`input()`** function reads the value entered by the user.

Syntax:

```
variable_name = input('Display message: ')
```

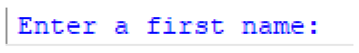
For example, if we want to collect a first name from the computer keyword and save it in variable **`msa`**:

```
msa=input('Enter a first name: ')
```

Once we have the information saved in variable **`msa`**, we can then prompt the value in a message as:

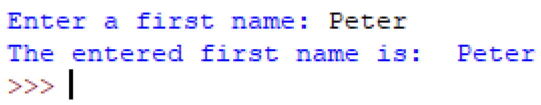
```
print('The entered first name is: ',msa)
```

Running the code, the command window will prompt the input message first:



```
Enter a first name:
```

If we type **Peter** and hit **Enter**, Python will move to the next code line and print the value of variable **`msa`**, which the name that the user typed and entered:



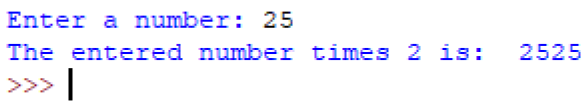
```
Enter a first name: Peter
The entered first name is:  Peter
>>> |
```

Python Casting

In computer programming, casting is a source code of a program that tells the compiler to generate the machine code that perform as the actual conversion. In other words, casting means to convert from one data type into another. For example, when we collect a number from the keyword, the value from the keyword is actually a string and not an integer or float type. Therefore, if we perform operations with the number collected from the keyword, if will perform a string operation and not a numerical operation. Let's take a look at the following code:

```
number1 = input("Enter a number: ")
product = number1 * 2
print("The entered number times 2 is: ", product)
```

Prompt result

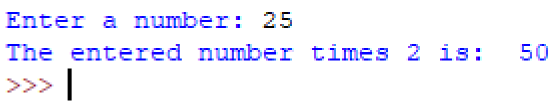


```
Enter a number: 25
The entered number times 2 is:  2525
>>> |
```

If we want to perform a numerical operation, then we need to cast the input type string to integer. To do so, we can use the Python casting *int( )* right after the input function:

```
number1 = input("Enter a number: ")
number1 = int(number1)
product = number1 * 2
print("The entered number times 2 is: ", product)
```

Prompt result



```
Enter a number: 25
The entered number times 2 is:  50
>>> |
```

or we can use Python casting within the **`input`** function

```
number1 = int(input("Enter a number: "))
product = number1 * 2
print("The entered number times 2 is: ", product)
```

Python Arithmetic Operators
In Python, we can do multiplication, addition, subtraction, division, return the remainder of a two numbers, exponentiation, integer division, and parentheses using the basic operators:

Python Arithmetic Operators	
Symbol	Operations
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
//	Integer division
**	Exponentiation
( )	Parentheses

Order of operation

Like a calculator, any programming language perform operations due to their order of operations. An operation is anything that uses an operator. For example, multiplication and division have a higher order than addition and subtraction. It basically means that if we have an equation that has multiplication and subtraction, the program will calculate the multiplication first and then the subtraction.

Example) if we solve for the following operation: 5 + 30 * 20, the operation will solve the multiplication first: 5 + 600, after the product, it will solve the addition giving 605.

Use of parentheses in a programming language to control the flow of the operations. Order of operation will always solve the equation inside the parentheses before proceeding with the basic operators. If we take 5 + 30 * 20 and add parentheses to it: (5 + 30) * 20, the operation will solve the operation inside the parentheses first, 35*20, after the sum, it will solve the multiplication giving 700

Parentheses can be nested, which means that there can be parentheses inside parentheses:

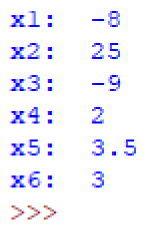
((5 + 30) * 20) / 10

In this case, the order of operation evaluates the innermost parentheses first: (35*20) / 10, then the outer parentheses: 700 / 10, and then the final division operator giving: 70.

Example) perform the following operations:

```
x1 = (6-7)*8
x2 = (3+2)**(6-4)
x3 = 3**2*-1
x4 = 5%3
x5 = 7/2
x6 = 7//2

print("x1: ", x1)
print("x2: ", x2)
print("x3: ", x3)
print("x4: ", x4)
print("x5: ", x5)
print("x6: ", x6)
```

Prompt result



```
x1:  -8
x2:  25
x3:  -9
x4:  2
x5:  3.5
x6:  3
>>>
```

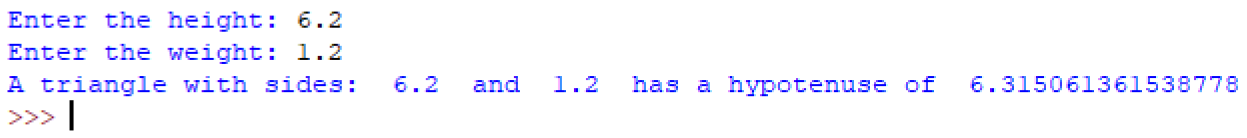
Example) ask the user to enter two sides of a right triangle as a float number and calculate the hypotenuse as

hyp = $(h^2+w^2)^{0.5}$

Prompt the two sides and the hypotenuse:

```
h = float(input("Enter the height: "))
w = float(input("Enter the width: "))
hyp = (h**2+w**2)**0.5
print("A triangle with sides: ", h, ",",w, " has a hypotenuse of ",hyp)
```

Prompt result



```
Enter the height: 6.2
Enter the weight: 1.2
A triangle with sides:  6.2  and  1.2  has a hypotenuse of  6.315061361538778
>>>
```

Python Assignment Operators

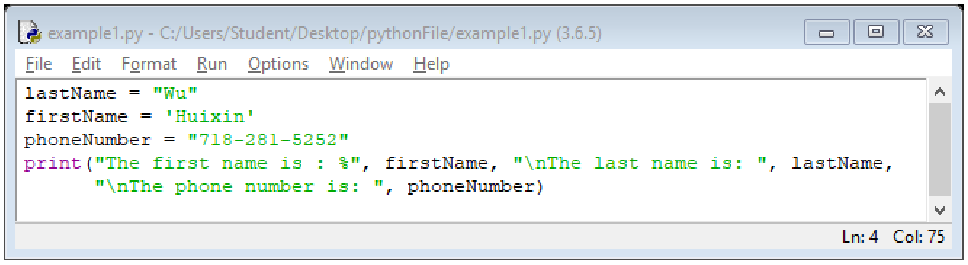
The Python assignment operators are used to assign values to the declared variables

Python Assignment Operators				
Symbol	**Operations**	**Python Example**	**Arithmetic equality**	**Value**
=	Assign a value	x = 5	x = 5	5
+=	Self-addition	x+=3	x = x+3	8
-=	Self-subtraction	x-=3	x = x-3	2
*=	Self-multiplication	x*=3	x = x*3	15
/=	Self-division	x/=3	x = x/3	$1.6\overline{6}$
//=	Self-division, returning only the quotient	x//=3	x = x//3	1
%=	Returning remainder of a self-division	x%=3	x = x%3	2
**=	Self-exponential	x**=3	x = x**3	125

Strings

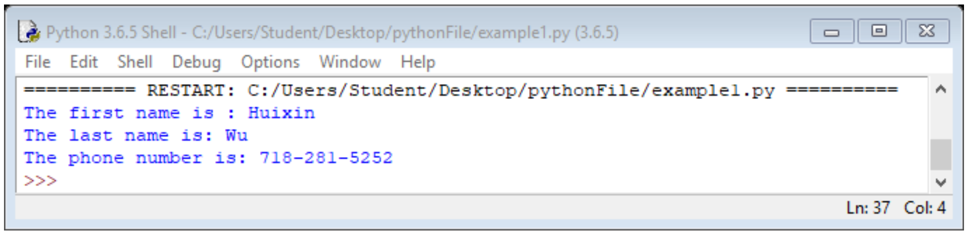
When programming, we usually call text a *string*. Think of a string as a collection of letters or characters. To create a string by putting single or double quotes around text because programming they need distinguish strings as value from variables. Also, we have to consistent in using single or double quotes, for example, if we start a string message with a single quote, we need to close the message with another single quote. If we start with a string with a single quote and we close it using a double quotes, when we run the Python program, the compiling will show us a syntax error.

Example) Use strings to create a complete message to display last name, first name, and phone number.


example1.py - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 75

```
lastName = "Wu"
firstName = 'Huixin'
phoneNumber = "718-281-5252"
print("The first name is : %", firstName, "\nThe last name is: ", lastName,
      "\nThe phone number is: ", phoneNumber)
```

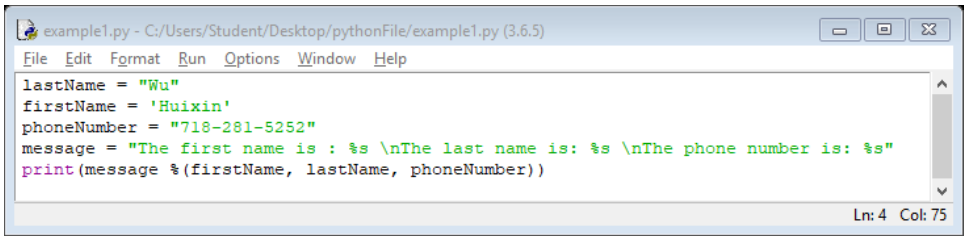
Result


Python 3.6.5 Shell - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Shell Debug Options Window Help
Ln: 37 Col: 4

```
========== RESTART: C:/Users/Student/Desktop/pythonFile/example1.py ==========
The first name is : Huixin
The last name is: Wu
The phone number is: 718-281-5252
>>>
```

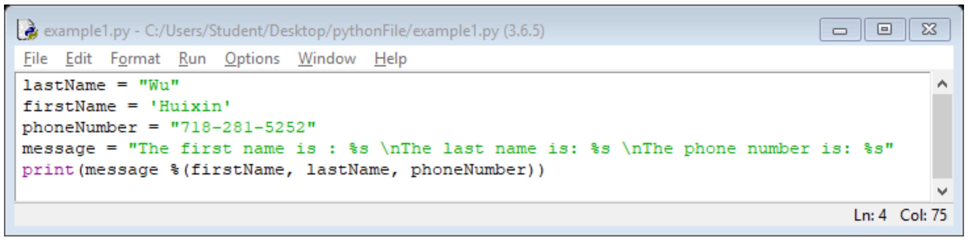
The comma symbol is used to concatenate to text with the variable.

We can also embed values in a string using **%s**, which is like a marker for a value that we want to add later.


example1.py - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 75

```
lastName = "Wu"
firstName = 'Huixin'
phoneNumber = "718-281-5252"
message = "The first name is : %s \nThe last name is: %s \nThe phone number is: %s"
print(message %(firstName, lastName, phoneNumber))
```

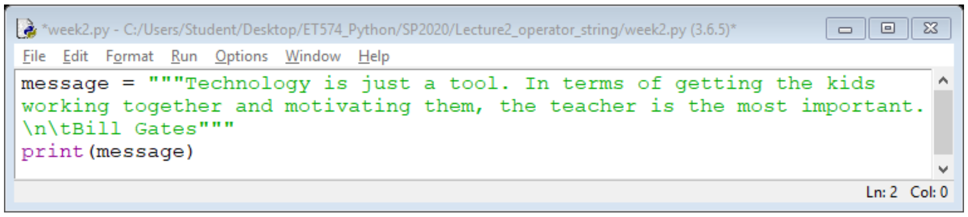
Result


example1.py - C:/Users/Student/Desktop/pythonFile/example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 75

```
lastName = "Wu"
firstName = 'Huixin'
phoneNumber = "718-281-5252"
message = "The first name is : %s \nThe last name is: %s \nThe phone number is: %s"
print(message %(firstName, lastName, phoneNumber))
```

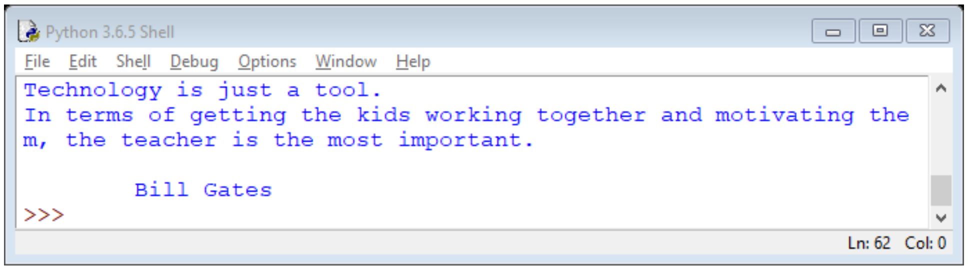
<u>Multiline Strings</u>

Triple double-quotes can use to create a multiline string to a variable


week2.py - C:/Users/Student/Desktop/ET574_Python/SP2020/Lecture2_operator_string/week2.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 0

```
message = """Technology is just a tool. In terms of getting the kids
working together and motivating them, the teacher is the most important.
\n\tBill Gates"""
print(message)
```

Result

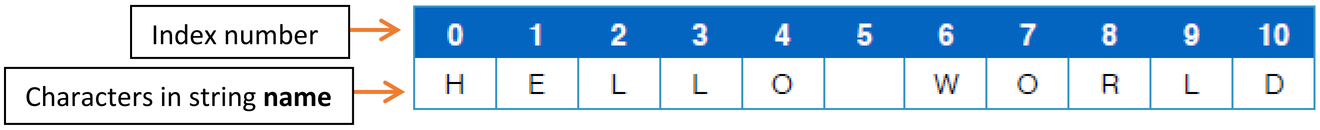


```
Technology is just a tool.
In terms of getting the kids working together and motivating the
m, the teacher is the most important.

        Bill Gates
>>>
```

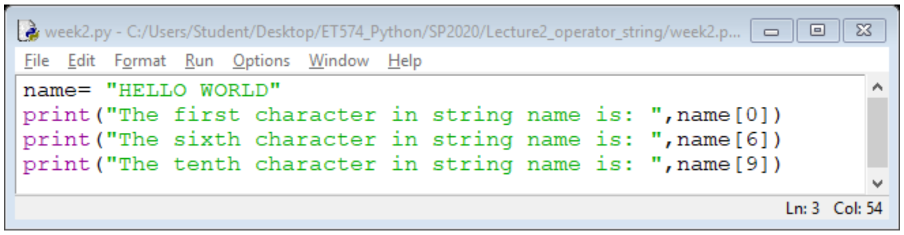
Strings are arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

The bracket operator `[]` can be used to read individual characters stored within a string. This requires an **index** value within the brackets (starting with 0) which indicates the character position in the string. We can think of string as an array of characters. So a string defined as the string **`name= “HELLO WORLD”`**



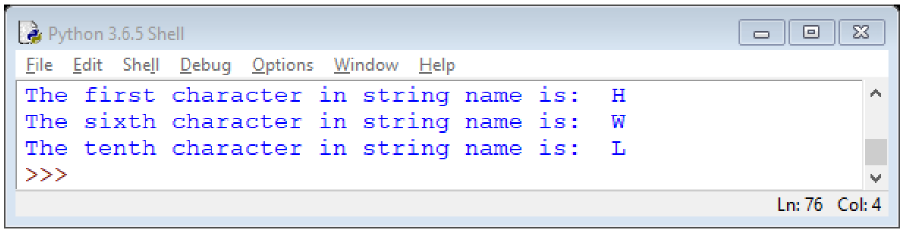
Index number	0	1	2	3	4	5	6	7	8	9	10
Characters in string **name**	H	E	L	L	O		W	O	R	L	D



week2.py - C:/Users/Student/Desktop/ET574_Python/SP2020/Lecture2_operator_string/week2.p...
File Edit Format Run Options Window Help
Ln: 3 Col: 54

```
name= "HELLO WORLD"
print("The first character in string name is: ",name[0])
print("The sixth character in string name is: ",name[6])
print("The tenth character in string name is: ",name[9])
```

Result



```
The first character in string name is:  H
The sixth character in string name is:  W
The tenth character in string name is:  L
>>>
```

String slices

We can set a range of characters of a string using the slice syntax. The slice syntax specify the start index, follow by the symbol colon **:** , and then the end index, without including the end index. A segment of a string is called **slice.**

Example) Print a segment of string **`strawberries`** from the second to the fifth character.

```
fruit = " strawberries"
print(fruit[1:5])
```

Result

```
traw
>>>
```

We can specific a segment from the beginning of a string up to end-index.

Example) print a segment of string **`strawberries`** from the beginning to the seventh character.

```
fruit = "strawberries"
print(fruit[:7])
```

Result

```
strawbe
>>>
```

On the other hand, we can also specific a segment from a starting-index to the last character of a string

Example) print a segment of string **`strawberries`** from the fifth character to last character.

```
fruit = "strawberries"
print(fruit[5:])
```

Result

```
berries
>>>
```

String length

Python uses a built-in function ***`Len()`*** that returns the number of characters in a string.

Example) find the length of string **`strawberries`**

```
fruit = "strawberries"
fruit_length = len(fruit)
print("The fruit has ", fruit_length, "characters")
```

Result

```
The fruit has 12 characters
>>>
```

We can also use the ***`Len()`*** to find the last character of our string. For example, to find the last word of **`color = blue`**

```
color = "blue"
color_len = len(color)
color_last_index = color_len - 1
```

```
color_last_character = color[color_last_index]
print("The last character in string:", color, "is",
color_last_character)
```

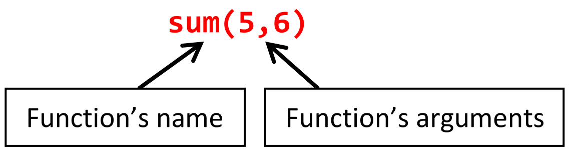
Result

```
The last character in string: blue is e
>>>
```

String Methods

Before we move on to how to use Python methods, let's review on some Python's terms which is function, classes/object, and methods. Almost everything in **Python** is an object, with its properties and methods. We are not going to have an intensive practice on the mentioned terms in this course, except for function, because they are standard features of Object Oriented Programming, OOP, which is the next level of computer programming.

Function: a function is a block of code to carry out a specific task and it is called by its name. All functions have arguments or not have arguments. The syntax of a function is as:



Classes: A **Class** is like an object constructor, or a "blueprint" for creating objects.

```
class MyClass:
  x = 5
```

Now we can use the class named **`MyClass`** to create objects. In this example, let's create an object named **`p1`** and print the value of **`x`**:

```
p1 = MyClass()
print(p1.x)
```

Object methods: Methods in objects are functions that belong to the object. A method is similar to a function—it takes arguments and returns a value—but the syntax is different.

Now that we now the basic terms in OOP, let's have a look at the built-in methods in Python for class **`string.`** Python has a set of built-in methods that we can use on strings:

- The **`strip`**`()` method removes any whitespace from the beginning or the end:

```
a = "    Hello, World!"
print(a.strip())
```

Result
```
    Hello, World!
    >>>
```

- The **lower()** method returns the string in lower case

```
a = "Hello, World!"
print(a.lower())
```

Result
```
    hello, world!
    >>>
```

- The **upper()** method returns the string in upper case

```
a = "Hello, World!"
print(a.upper())
```

Result
```
    HELLO, WORLD!
    >>>
```

- The **replace()** method replaces a string with another string

```
a = "Hello, World!"
print(a.replace("H", “J”))
```

Result
```
    Jello, World!
    >>>
```

- The **split()** method splits the string into substrings if it finds instances of the separator

```
a = "Hello World! Welcome to Python"
print(a.split("!"))
```

Result
```
    ['Hello World', ' Welcome to Python']
    >>>
```

ADD SECOND ARGUMENT FOR split() method

- The **find()** method finds the character within the parenthesis

```
a = "Hello World! "
index = a.find("o")
```

```
print("index for letter o is:",index)
```

Result

```
index for letter o is: 4
>>>
```

We can also find a character of a string after an index:

```
a = "Hello World! "
index = a.find("o",5)
print("index for letter o is:",index)
```

Result

```
index for letter o is: 7
>>>
```

The **in** operator

Python uses membership operators **in** and **not in** is used to check if a certain phrase or character is present in a string. The word **in** is a Boolean operator that takes two strings and returns ***True*** if the first appears as a substring in the second.

```
msg = "Hello World! "
answer = "e" in msg
print("Is character e in string?",answer)
```

Result

```
Is character e in string? True
>>>
```

```
msg = "Hello World! "
answer = "e" not in msg
print("Is character e in string?",answer)
```

Result

```
Is character e in string? False
>>>
```

Concatenate Strings

To concatenate or combine two or more strings, we can use the + operator

```
a = "apple"
b = ", "
c = "grapes"
```

```
message = c+b+a+b+c
print(message)
```

Result

```
      grapes, apple, grapes
      >>>
```

String `format()`

Python cannot combine strings and integers or float value using the + operator. Instead, Python uses the **`format()`** method to combine string and numbers. The **`format()`** method takes the passed arguments, formats them, and places them in the string where the placeholders **`{}`** are:

```
age = 36
txt = "My name is John, and I am {}"
new_txt=txt.format(age)
print(new_txt)
```

Result

```
      My name is John, and I am 36
      >>>
```

We can also use multiple placeholder

```
price = 49.95
quantity = 3
itemno = 567
myorder = "I want {} pieces of item #{} for {} dollars."
final_order = myorder.format(quantity, itemno, price)
print(final_order)
```

Result

```
      I want 3 pieces of item #567 for 49.95 dollars.
```

You can use index numbers **`{0}`** to be sure the arguments are placed in the correct placeholders:

```
price = 49.95
quantity = 3
itemno = 567
myorder = "I want to pay {2} dollars for {0} pieces of item #{1}."
print(myorder.format(quantity, itemno, price))
```

Result

```
      I want to pay 49.95 dollars for 3 pieces of item #567.
```

List

List is the same as array. We can store various values within the same variable name. Those values are organized by an index location. The index location starts from zero.

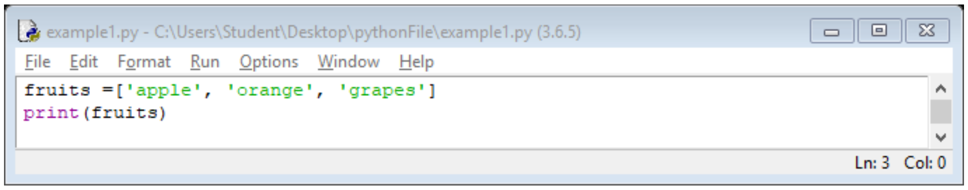
To create a list by using square brackets [ ]

fruits =['apple', 'orange', 'grapes']

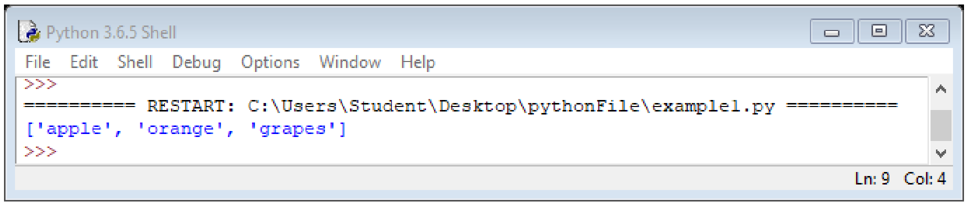
The lists *fruits* will be organized as the following:

fruits =	apple	orange	grapes
Index location	0	1	2

The Python code will look as the following:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 3 Col: 0

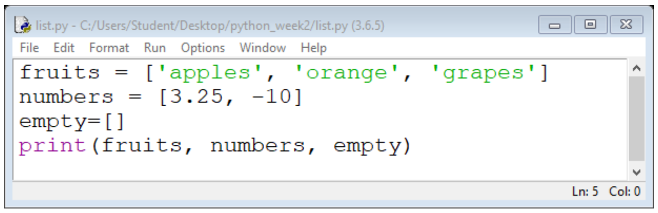
```
fruits =['apple', 'orange', 'grapes']
print(fruits)
```



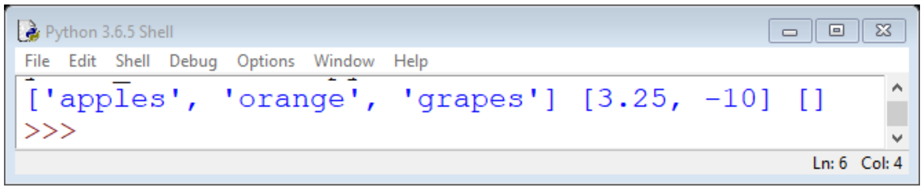
```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
['apple', 'orange', 'grapes']
>>>
```

<u>Empty list</u>

A list that contains no elements is called an empty list; you can create one with empty brackets, **[ ]**


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 5 Col: 0

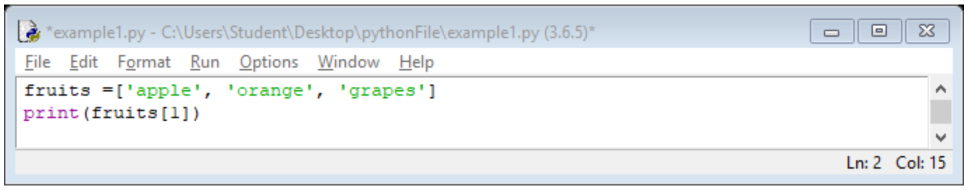
```
fruits = ['apples', 'orange', 'grapes']
numbers = [3.25, -10]
empty=[]
print(fruits, numbers, empty)
```



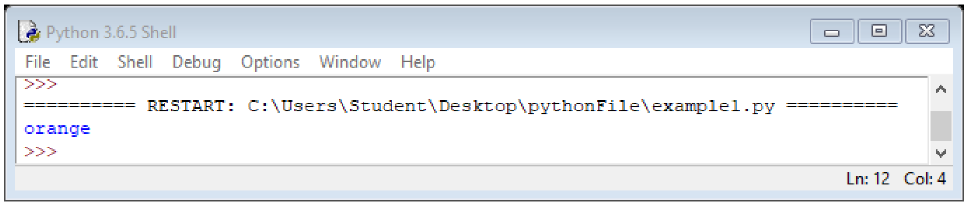
```
['apples', 'orange', 'grapes'] [3.25, -10] []
>>>
```

Accessing to elements in the list

Creating a list takes a bit more typing than creating a string, but a list is more useful than a string because it can be manipulated. The syntax for accessing the elements of a list is the same as for accessing the characters of a string — the **bracket operator [ ]**. For example, we could print the second value in *fruits* list by entering its index position inside square brackets after the list name. Remember that the index starts from zero, therefore, if we want to print the second value, the index number should be 1 **`fruits[1]`**

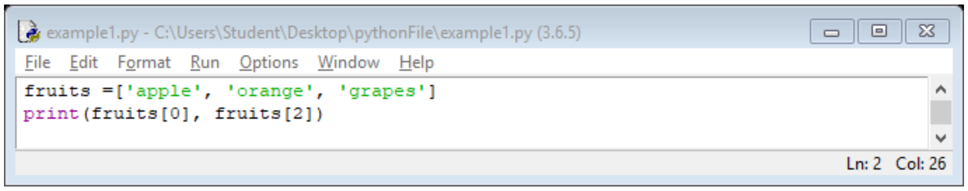

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 15

```
fruits =['apple', 'orange', 'grapes']
print(fruits[1])
```

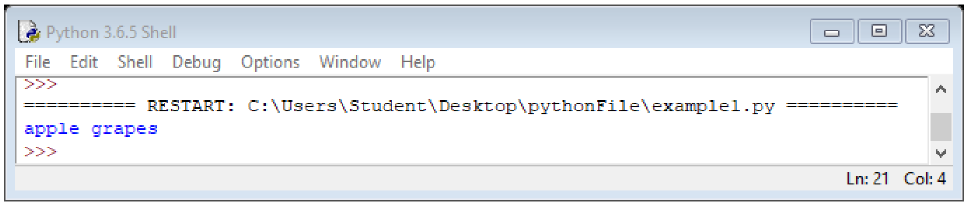


```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
orange
>>>
```

Also, if we want to print the first and third values of list *fruits*, we need to separate the index number with a comma:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 26

```
fruits =['apple', 'orange', 'grapes']
print(fruits[0], fruits[2])
```

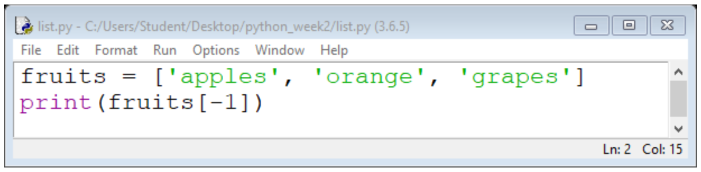


```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
apple grapes
>>>
```

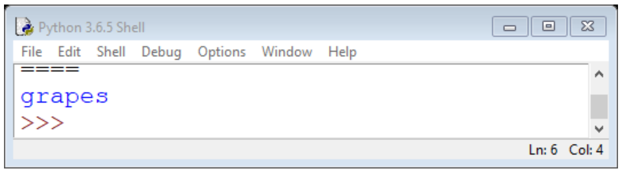
Negative index

If an index has a negative value, it counts backward from the end of the list.

fruits =	apple	orange	grapes
Index location	-3	-2	-1


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 15

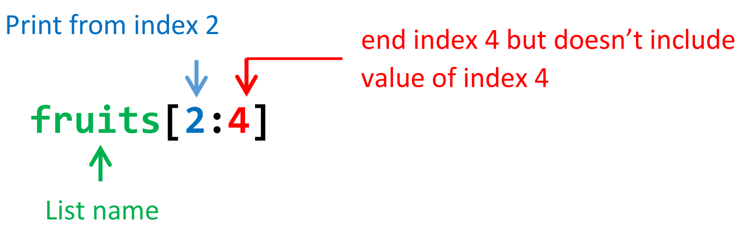
```
fruits = ['apples', 'orange', 'grapes']
print(fruits[-1])
```



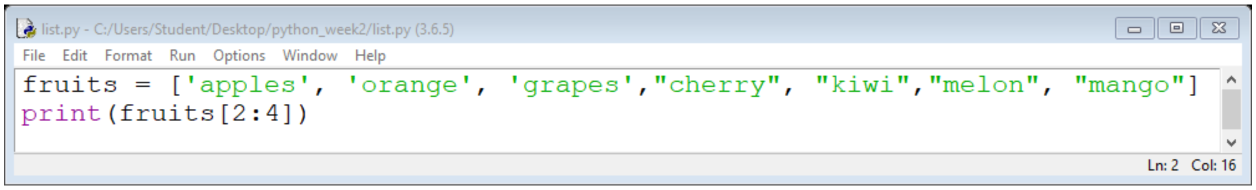
```
====
grapes
>>>
```

Range of indexes

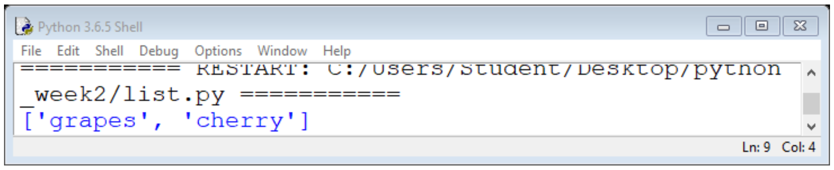
You can specify a range of indexes by specifying where to start and where to end the range by using a colon operator. Remember that Python will not print the end index



Example) Range of indexes: print list elements from index 2 up to index 4 (Exclusive)

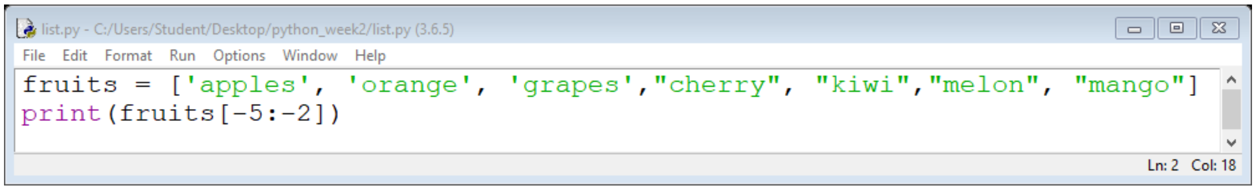

list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 16

```
fruits = ['apples', 'orange', 'grapes',"cherry", "kiwi","melon", "mango"]
print(fruits[2:4])
```

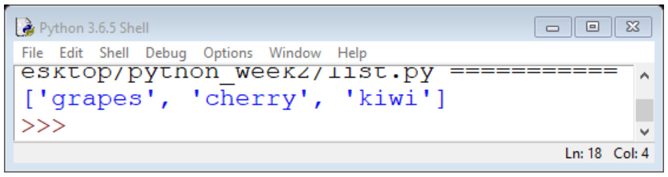


```
=========== RESTART: C:/Users/Student/Desktop/python
_week2/list.py ===========
['grapes', 'cherry']
```

Example) Negative range of indexes: print list elements from element -5 up to element -2


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 18

```
fruits = ['apples', 'orange', 'grapes',"cherry", "kiwi","melon", "mango"]
print(fruits[-5:-2])
```



```
esktop/python_week2/list.py ===========
['grapes', 'cherry', 'kiwi']
>>>
```

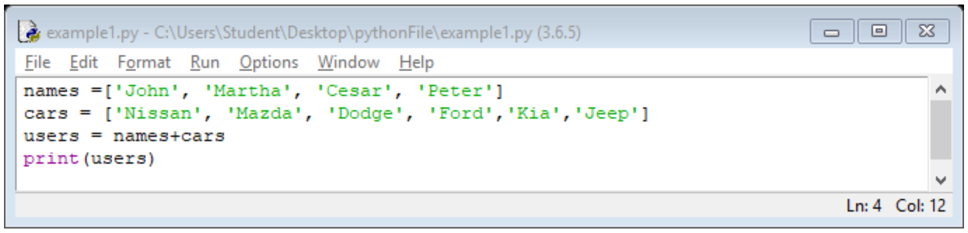
List operations

+ operator

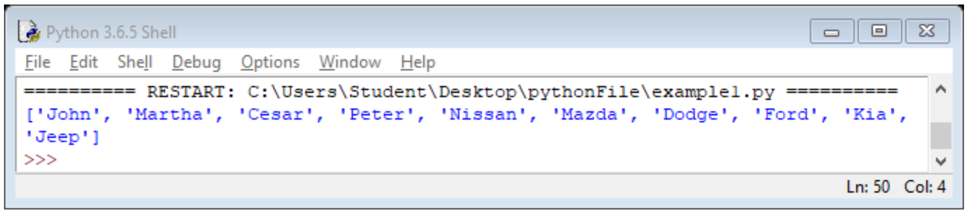
The + operator concatenates lists. For example, suppose we have two lists: names and cars:

```
names =['John', 'Martha', 'Cesar', 'Peter']

cars = ['Nissan', 'Mazda', 'Dodge', 'Ford','Kia','Jeep']
```

We can also add the two lists and set the result equal to another variable:

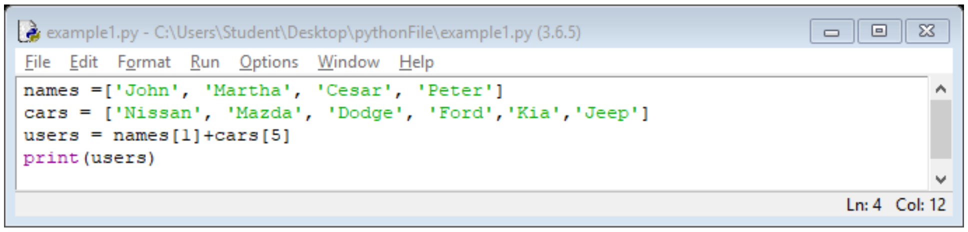

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 12

```
names =['John', 'Martha', 'Cesar', 'Peter']
cars = ['Nissan', 'Mazda', 'Dodge', 'Ford','Kia','Jeep']
users = names+cars
print(users)
```

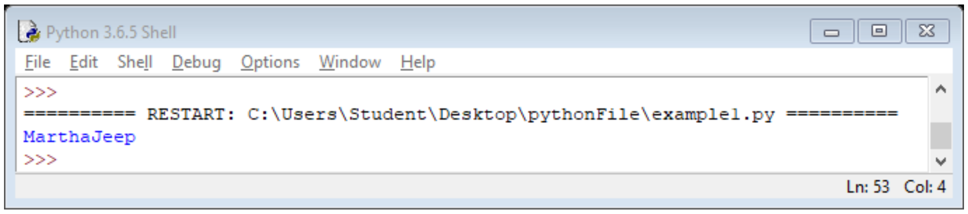


```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
['John', 'Martha', 'Cesar', 'Peter', 'Nissan', 'Mazda', 'Dodge', 'Ford', 'Kia',
'Jeep']
>>>
```

We can also add independent values from different lists. For example, if we want to join Martha with Jeep: users = **`names[1]+cars[5]`**


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 12

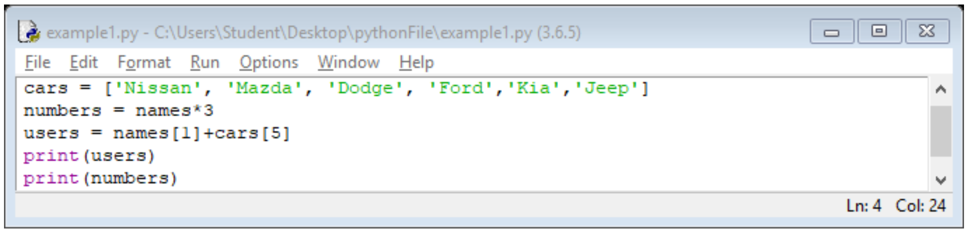
```
names =['John', 'Martha', 'Cesar', 'Peter']
cars = ['Nissan', 'Mazda', 'Dodge', 'Ford','Kia','Jeep']
users = names[1]+cars[5]
print(users)
```



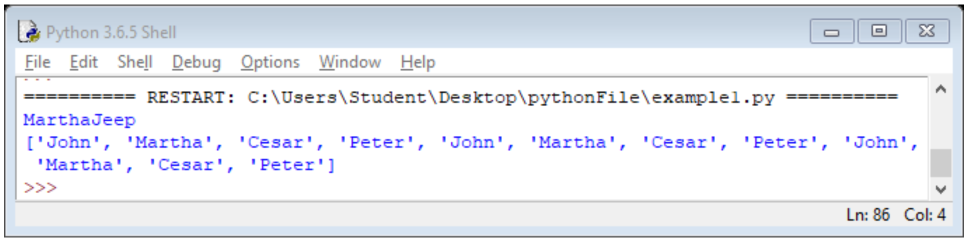
```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
MarthaJeep
>>>
```

** operator*

Similarly, the * operator repeats a list a given number of times. In other words, we can also duplicate list values by using the multiplication sign *


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 24

```
cars = ['Nissan', 'Mazda', 'Dodge', 'Ford','Kia','Jeep']
numbers = names*3
users = names[1]+cars[5]
print(users)
print(numbers)
```

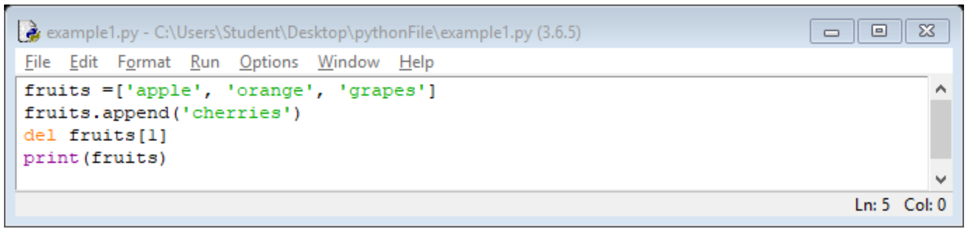


```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
MarthaJeep
['John', 'Martha', 'Cesar', 'Peter', 'John', 'Martha', 'Cesar', 'Peter', 'John',
 'Martha', 'Cesar', 'Peter']
>>>
```

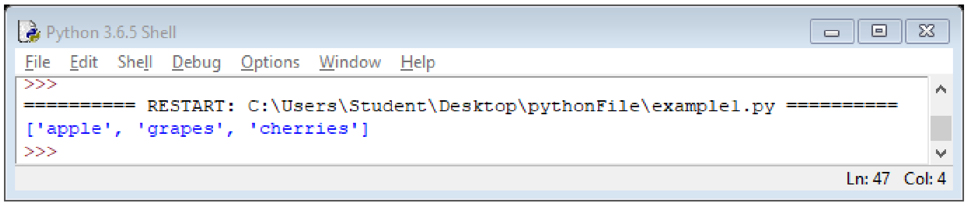
List Methods

Removing Items from a list

To remove items from a list, use the *del* command (short for *delete*) follows by the list value. For example, to remove the second value from list *fruits*: `del fruits[1]`


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 5 Col: 0

```
fruits =['apple', 'orange', 'grapes']
fruits.append('cherries')
del fruits[1]
print(fruits)
```

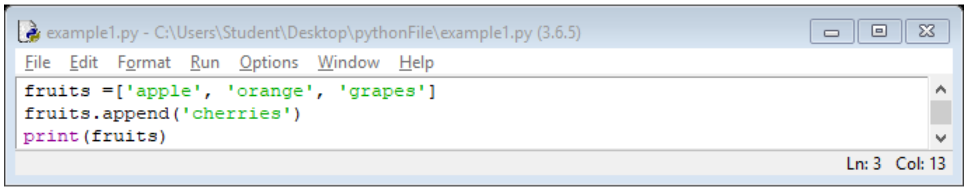


```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
['apple', 'grapes', 'cherries']
>>>
```

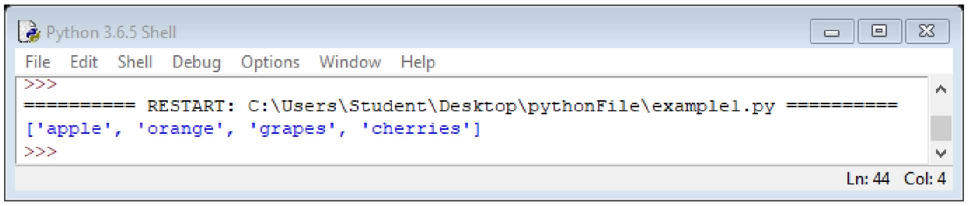
Adding items to a lists

To add items to a list, we use the Python function ***append( ).*** A function is a chuck of code that tells Python to do something. In this case, ***append( )*** adds an item to the end of a list.

For example, to add *cherries* to the list *fruits*: `fruits.append('cherries')`



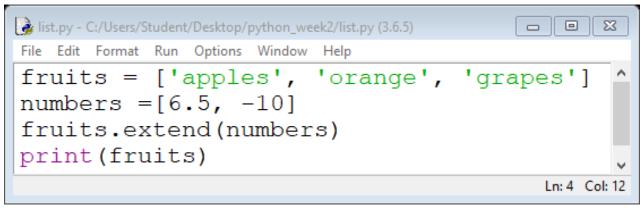
```
fruits =['apple', 'orange', 'grapes']
fruits.append('cherries')
print(fruits)
```



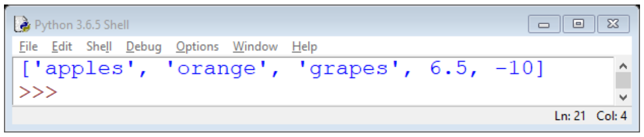
```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
['apple', 'orange', 'grapes', 'cherries']
>>>
```

Extend item in a list

You can use the **`extend()`** method, which purpose is to add elements from one list to another list:



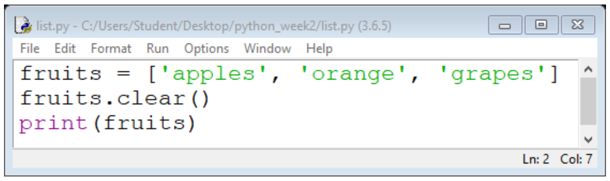
```
fruits = ['apples', 'orange', 'grapes']
numbers =[6.5, -10]
fruits.extend(numbers)
print(fruits)
```



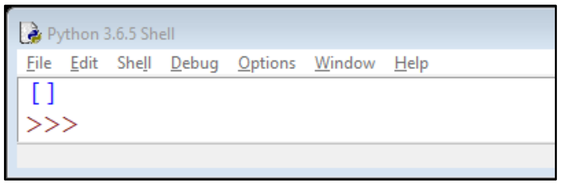
```
['apples', 'orange', 'grapes', 6.5, -10]
>>>
```

Clear elements from a list

Method **`clear()`** Removes all the elements from the list



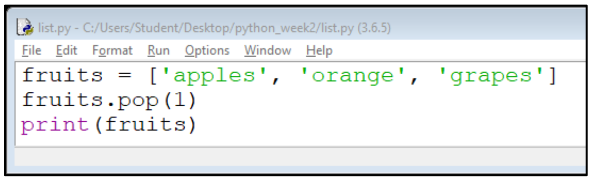
```
fruits = ['apples', 'orange', 'grapes']
fruits.clear()
print(fruits)
```



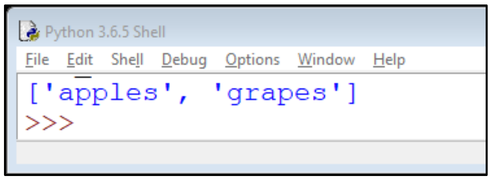
```
[]
>>>
```

Remove an element from the list with specific index

The **`pop()`** method removes the element at the specified position. Method that returns a value



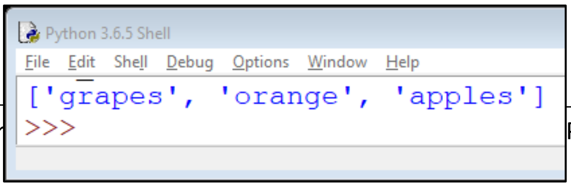
```
fruits = ['apples', 'orange', 'grapes']
fruits.pop(1)
print(fruits)
```



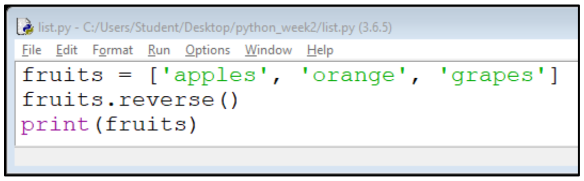
```
['apples', 'grapes']
>>>
```

Reverse a list

The **`reverse()`** method reverses the sorting order of the elements.



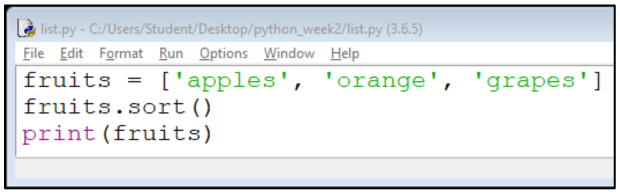
```
['grapes', 'orange', 'apples']
>>>
```


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help

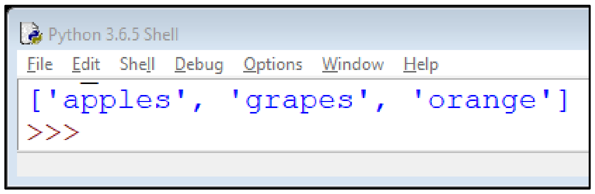
```
fruits = ['apples', 'orange', 'grapes']
fruits.reverse()
print(fruits)
```

Sort elements in an array

The `sort()` method sorts the list ascending by default. If it is string list, it sorts alphabetically

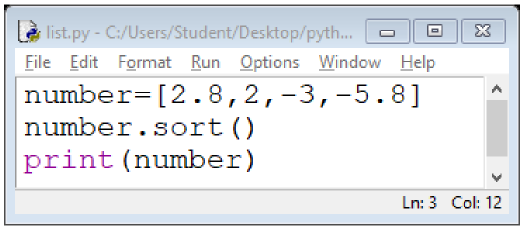

list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help

```
fruits = ['apples', 'orange', 'grapes']
fruits.sort()
print(fruits)
```

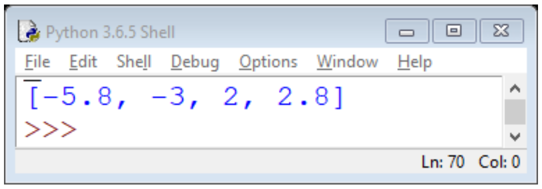


```
['apples', 'grapes', 'orange']
>>>
```

If it is number list, it sorts by increasing order.


list.py - C:/Users/Student/Desktop/pyth...
File Edit Format Run Options Window Help
Ln: 3 Col: 12

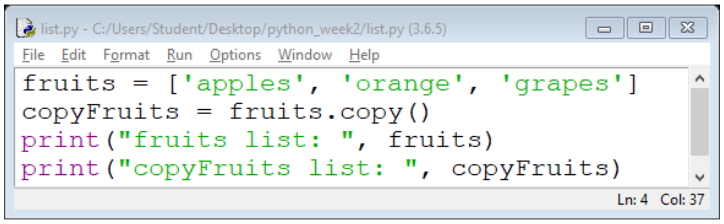
```
number=[2.8,2,-3,-5.8]
number.sort()
print(number)
```



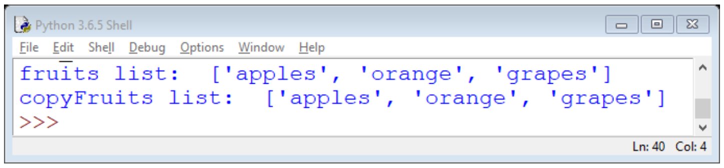
```
[-5.8, -3, 2, 2.8]
>>>
```

copy a list

`copy()` returns a copy of the list


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 37

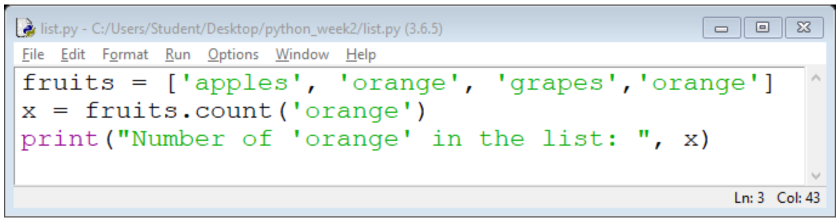
```
fruits = ['apples', 'orange', 'grapes']
copyFruits = fruits.copy()
print("fruits list: ", fruits)
print("copyFruits list: ", copyFruits)
```



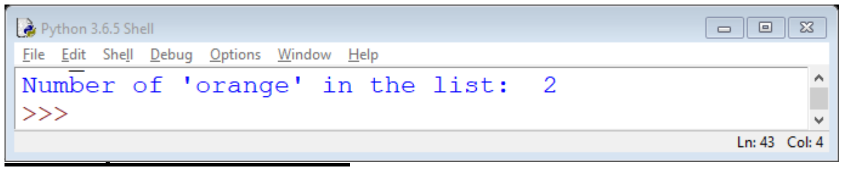
```
fruits list:  ['apples', 'orange', 'grapes']
copyFruits list:  ['apples', 'orange', 'grapes']
>>>
```

Return number of a specific value

The `count()` method **returns** the number of elements with the specified value.

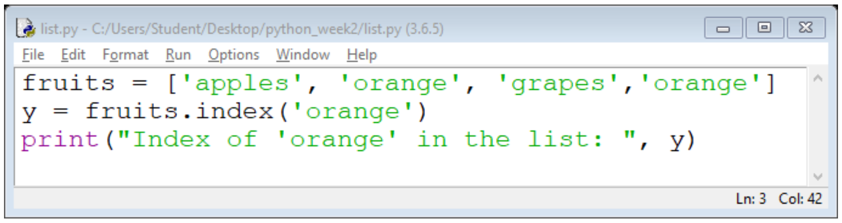

list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 3 Col: 43

```
fruits = ['apples', 'orange', 'grapes','orange']
x = fruits.count('orange')
print("Number of 'orange' in the list: ", x)
```

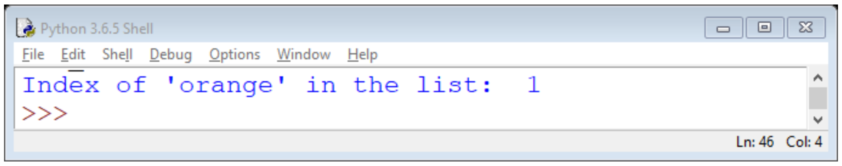


```
Number of 'orange' in the list:  2
>>>
```

The `index()` method returns the position at the first occurrence of the specified value.


list.py - C:/Users/Student/Desktop/python_week2/list.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 3 Col: 42

```
fruits = ['apples', 'orange', 'grapes','orange']
y = fruits.index('orange')
print("Index of 'orange' in the list: ", y)
```



```
Index of 'orange' in the list:  1
>>>
```

If and else statement

Asking question using if and else statement

In programming, we often ask yes or no questions, and decide to do something based on the answer. For example, we might ask, "Are you older than 21?" and if the answer is yes, respond with "You are an adult!"

Those sorts of questions are called *conditions*, and we combine these conditions and the responses into *if* statement. Conditions can be more complicated than a single question, and *if* statements can also be combined with multiple questions and different responses based on the answer to each questions.

Conditions help us compare things

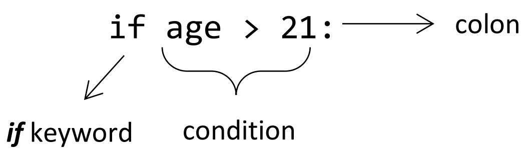
A condition is a programing statement that compares things and tells us whether the criteria set by the comparison are either True of False.

We use symbols is Python, called *comparison operators*, to create our conditions:

Python Comparison Operators	
Symbol	Definition
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

if statement

An *if* statement is a conditional statement that check if the statement is true. The *if* statement in Python is made up of the *if* keyword, followed by a condition and a colon:

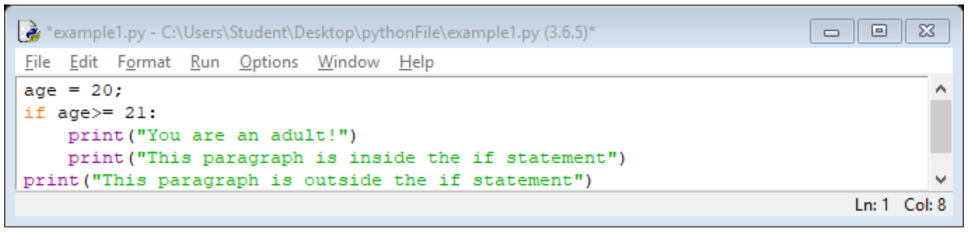


```
if age > 21:
```

After the colon, the next line should be command blocks, which will run if the conditional is True, otherwise, the program will skip the entire command blocks. Python identifies command blocks with a tab (inserted when you press the TAB key) or four spaces at the beginning from left to right:

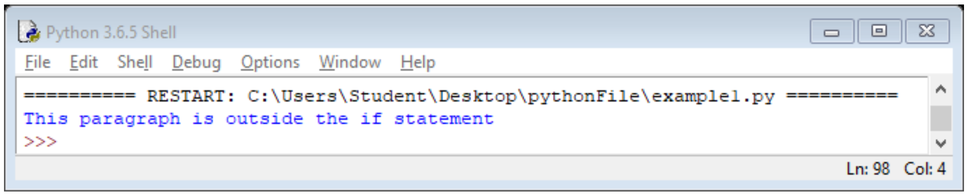
```
if age>= 21:
    print("You are an adult!")
```
← Command block

For example, if we assign value 20 to variable *age* in the following program:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 1 Col: 8

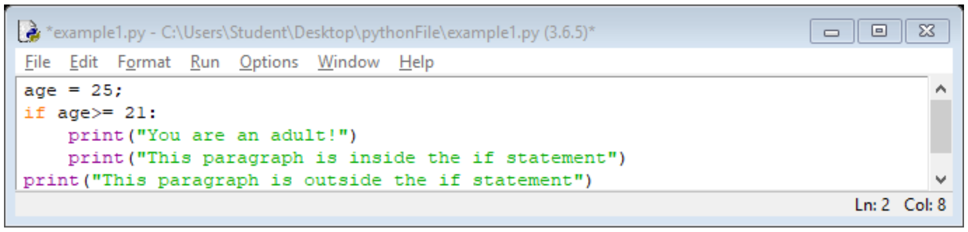
```
age = 20;
if age>= 21:
    print("You are an adult!")
    print("This paragraph is inside the if statement")
print("This paragraph is outside the if statement")
```

Since the statement is False, what is inside the *if* statement will not display. Then it will run only the line after the *if* statement

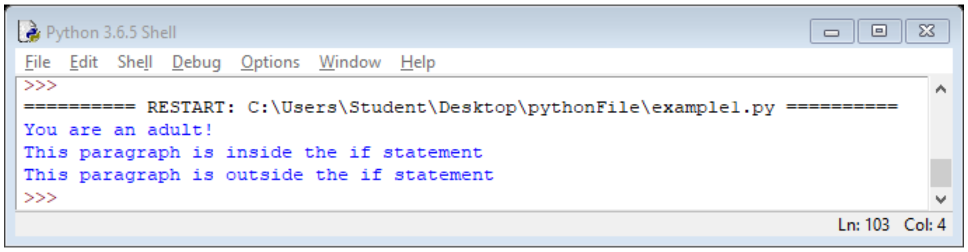


```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
This paragraph is outside the if statement
>>>
```

If we assign the value 25 to variable *age*, the statement will be true, the command blocks inside the *if* statement will run, and then the lines after the *if* statement.


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 8

```
age = 25;
if age>= 21:
    print("You are an adult!")
    print("This paragraph is inside the if statement")
print("This paragraph is outside the if statement")
```



```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
You are an adult!
This paragraph is inside the if statement
This paragraph is outside the if statement
>>>
```

In Python, *whitespace*, is meaningful. Code that is at the same position (indented the same number of spaces from the left margin) is grouped into a block, and whenever you start a new line with more spaces than the previous one:

```
Block 1 = line of code2
Block 1 = line of code3
    Block 2 = line of code4 (inside of code3)
    Block 2 = line of code5 (inside of code3)
```

```
                Block 3 = line of code6 (inside of code5)
        Block 2 = line of code7 (inside of code3)
Block 1 = line of code8

        Block 4 = line of code9 (inside of code8)
```

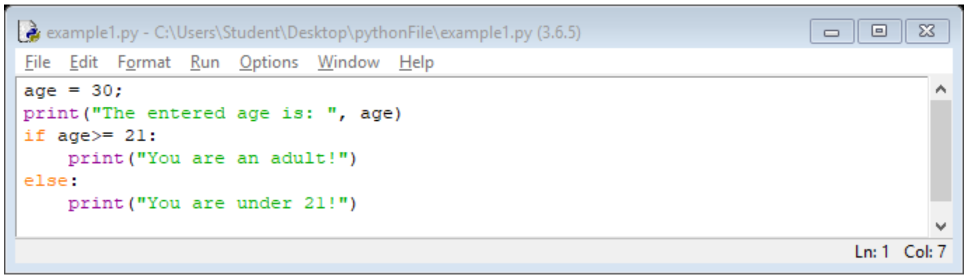
We group statements together into blocks because they are related. The statements need to be run together. When we change the indentation, we are generally creating new blocks.

From the previous blocks of line, even though Block 2 and Block 4 have the same indentation, they are considered different blocks because Block 2 is inside of code 3 and Block 4 is inside of code 8.

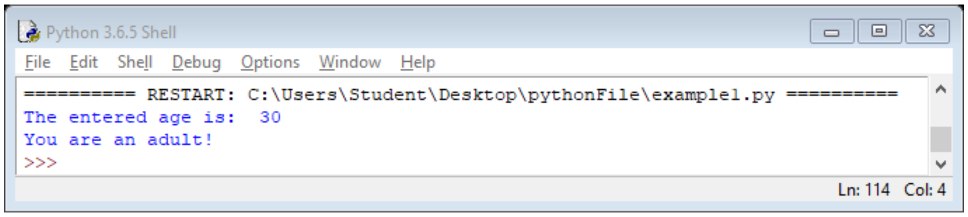
If – else statement

In addition to use *if* statements to do something when a condition is True, we can also use *if* statements to do something when a condition is not true. For example, we might print one message to the screen if your age is 21 (True) and another if your age is not 21 (False). The trick here is to use an *if-else* statement, which essentially says "if the input age is greater than or equal to 21, then print: You are an adult!; or else, print: You are under 21!"

For example, if we enter age = 30, then *if* statement is true, therefore it will run the block inside the *if* statement:

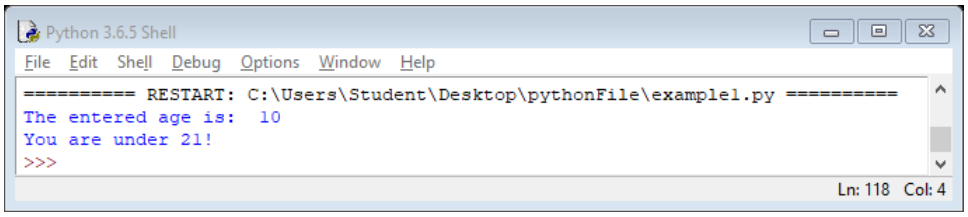

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 1 Col: 7

```
age = 30;
print("The entered age is: ", age)
if age>= 21:
    print("You are an adult!")
else:
    print("You are under 21!")
```



```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
The entered age is:  30
You are an adult!
>>>
```

Otherwise, if you enter age = 10, then *if* statement is false, therefore it will run the block inside the *else* statement:



```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
The entered age is:  10
You are under 21!
>>>
```

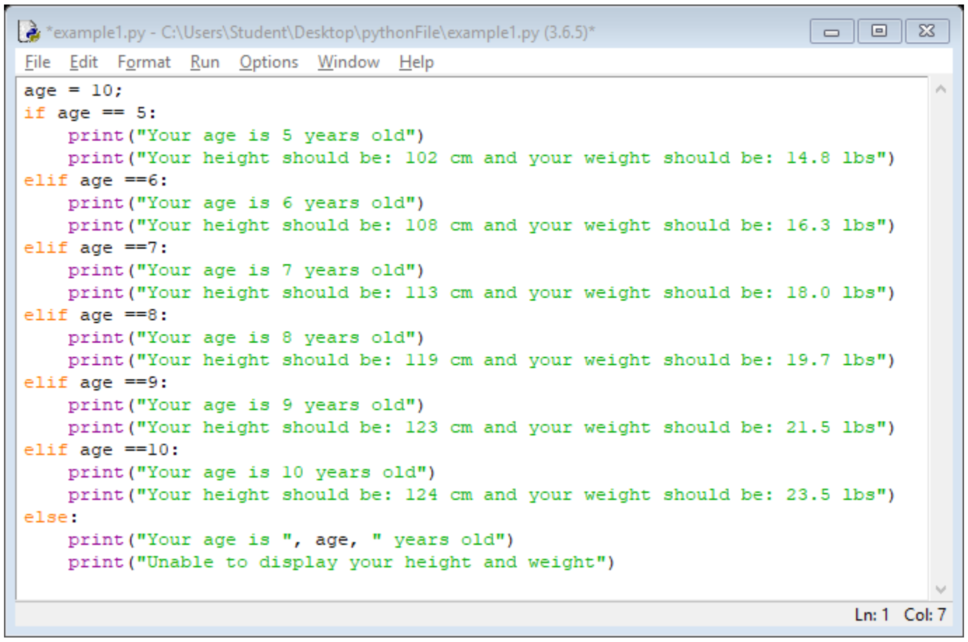
if-elif-else statement

We can extend an *if* statement even further with *elif*, which is a short for else-if. For example, we can check if a person's age is 10, 11, 12, and so on.

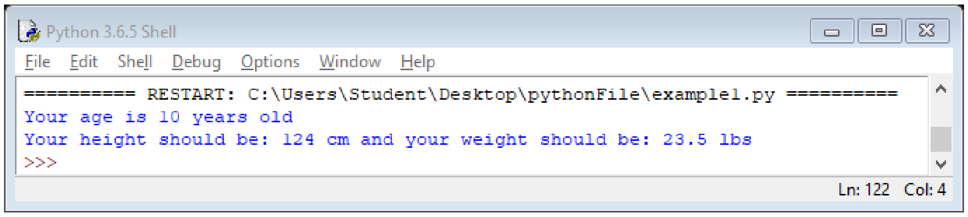
Example) check the age between 5 and 10. For each age, print their average height and weight using the following table:

Age	Height	Weight
5	102	14.8
6	108	16.3
7	113	18.0
8	119	19.7
9	123	21.5
10	124	23.5

If the age is not between 5 and 10, then print: *Unable to display your height and weight*. For example, if we set the age value to 10:

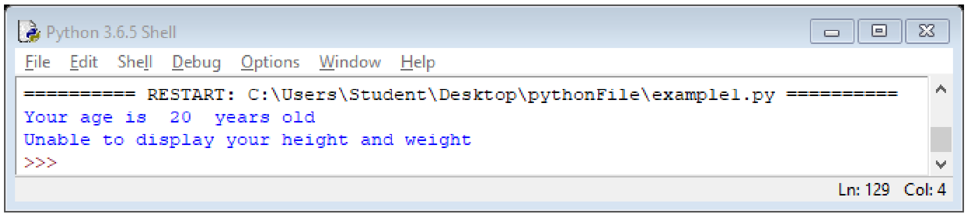

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 1 Col: 7

```
age = 10;
if age == 5:
    print("Your age is 5 years old")
    print("Your height should be: 102 cm and your weight should be: 14.8 lbs")
elif age ==6:
    print("Your age is 6 years old")
    print("Your height should be: 108 cm and your weight should be: 16.3 lbs")
elif age ==7:
    print("Your age is 7 years old")
    print("Your height should be: 113 cm and your weight should be: 18.0 lbs")
elif age ==8:
    print("Your age is 8 years old")
    print("Your height should be: 119 cm and your weight should be: 19.7 lbs")
elif age ==9:
    print("Your age is 9 years old")
    print("Your height should be: 123 cm and your weight should be: 21.5 lbs")
elif age ==10:
    print("Your age is 10 years old")
    print("Your height should be: 124 cm and your weight should be: 23.5 lbs")
else:
    print("Your age is ", age, " years old")
    print("Unable to display your height and weight")
```



```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
Your age is 10 years old
Your height should be: 124 cm and your weight should be: 23.5 lbs
>>>
```

If we set age = 20:



```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
Your age is  20  years old
Unable to display your height and weight
>>>
```

Combining Conditions

We can combine conditions by using the keywords ***and*** and ***or*** logical operator.

Operator	Description	Example
and	Returns TRUE if both statements are TRUE	age > 5 **and** age <10
or	Returns TRUE if one of the statements is TRUE	age == 5 **or** age == 6
not	Reserve the result, returns FALSE if the result is TRUE	**not**(age > 5 **and** age <10)

The and operator

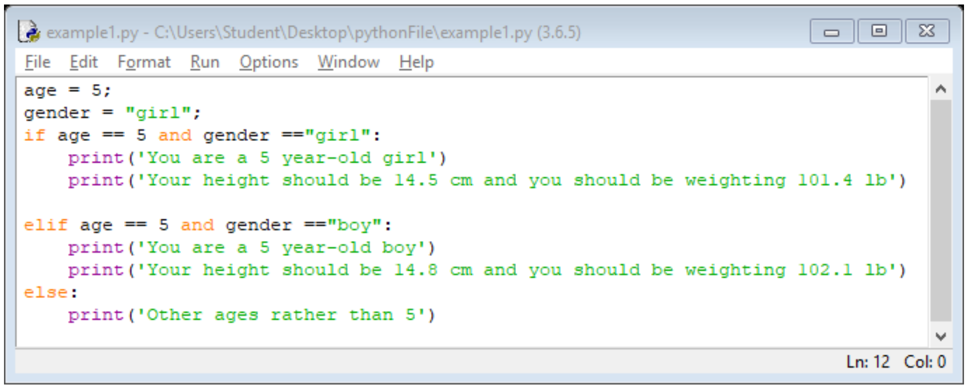
The *and* operator is used to indicate that the statement will be true when all the conditions are true. For example, if we want to print that if the kid is a 5 year-old girl, her weight and height should 14.5cm and 101.4 lb respectively:

```
if age == 5 and gender =="girl" :
```

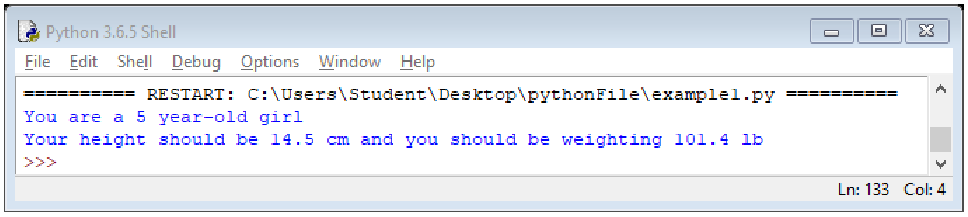
We can also add to the code that if the kid as a 5 year-old boy, his weight and height should 14.8 cm and 102.1 lb respectively:

```
elif age == 5 and gender =="boy":
```

The complete code will be as the following:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 12 Col: 0

```
age = 5;
gender = "girl";
if age == 5 and gender =="girl":
    print('You are a 5 year-old girl')
    print('Your height should be 14.5 cm and you should be weighting 101.4 lb')

elif age == 5 and gender =="boy":
    print('You are a 5 year-old boy')
    print('Your height should be 14.8 cm and you should be weighting 102.1 lb')
else:
    print('Other ages rather than 5')
```



```
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
You are a 5 year-old girl
Your height should be 14.5 cm and you should be weighting 101.4 lb
>>>
```

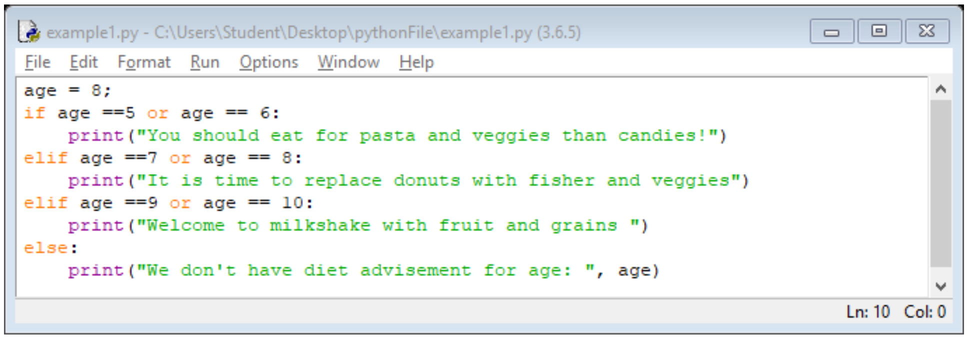
The ***or*** operator is used to indicate that the statement will be true when at least one of the condition is true. For example, if a kid is between the age of 5 and 6, it will print: *`You should eat for pasta and veggies than candies!`*

```
if age ==5 or age == 6:
    print("You should eat more pasta and cheese than candies!")
```

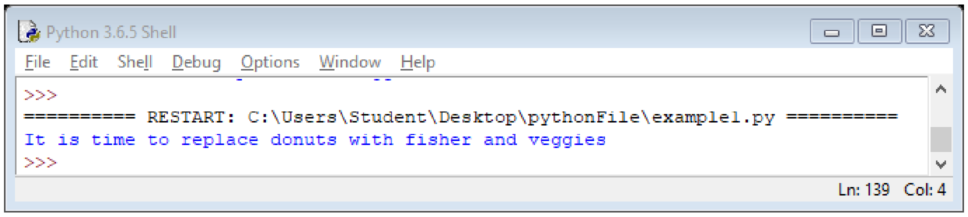
We can create a code using logical conditions as following:

Ages	Diet message
5 and 6	You should eat more pasta and cheese than candies
7 and 8	It is time to replace donuts with fisher and veggies
9 and 10	Welcome to milkshake with fruit and grains

The complete code will look like:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 10 Col: 0

```
age = 8;
if age ==5 or age == 6:
    print("You should eat for pasta and veggies than candies!")
elif age ==7 or age == 8:
    print("It is time to replace donuts with fisher and veggies")
elif age ==9 or age == 10:
    print("Welcome to milkshake with fruit and grains ")
else:
    print("We don't have diet advisement for age: ", age)
```



```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
It is time to replace donuts with fisher and veggies
>>>
```

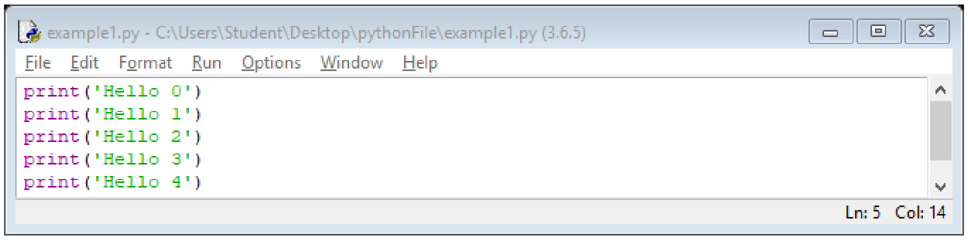
Going Loopy

Nothing is worse than having to do the same thing over and over again. Programmers don't particularly like repeating codes. Therefore, programming languages have what is called *loops*, which allow programmers to set the same code to repeat as many as needed, even to set a loop to run infinite time. The most common loops are *for* and *while* loops.

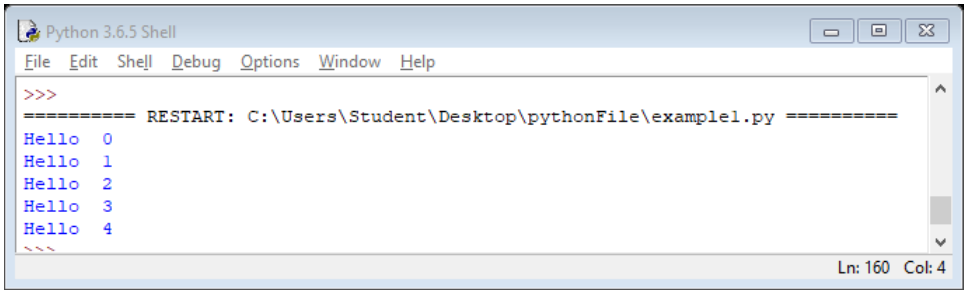
for loops

A for loop is used for iterating over a sequence. With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

For example, to print hello five times, we could like the following lines of code:

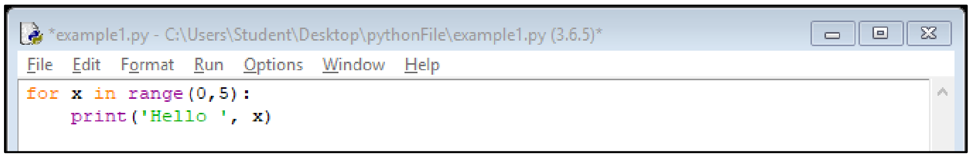

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 5 Col: 14

```
print('Hello 0')
print('Hello 1')
print('Hello 2')
print('Hello 3')
print('Hello 4')
```



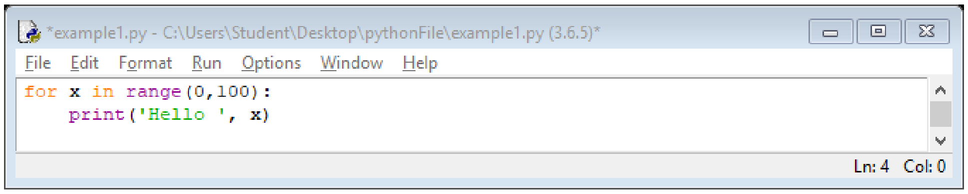
```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
>>>
```

Now, think in what should we do if we need to print the same hello line 100 times? Should we write 100 code of lines? Instead, we can use a *for* loop to reduce the amount of typing and repetition:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help

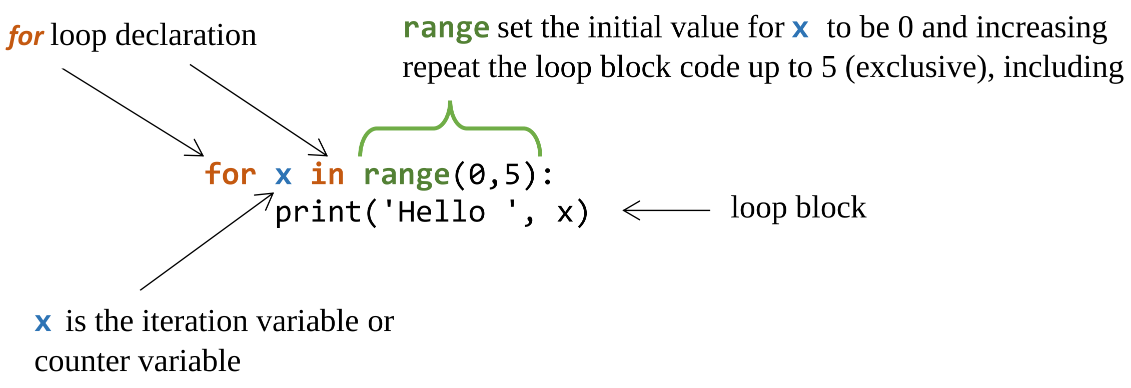
```
for x in range(0,5):
    print('Hello ', x)
```

For instant, to repeat the same hello line 100 times, we should only change the range:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 4 Col: 0

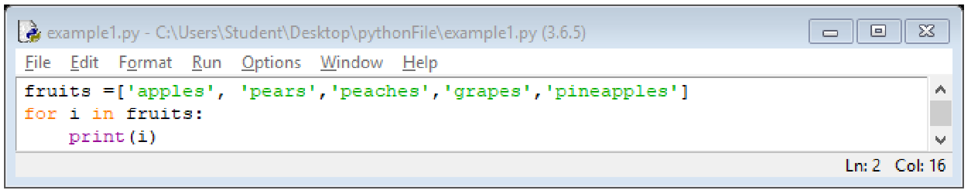
```
for x in range(0,100):
    print('Hello ', x)
```

The *for* loop syntax is:

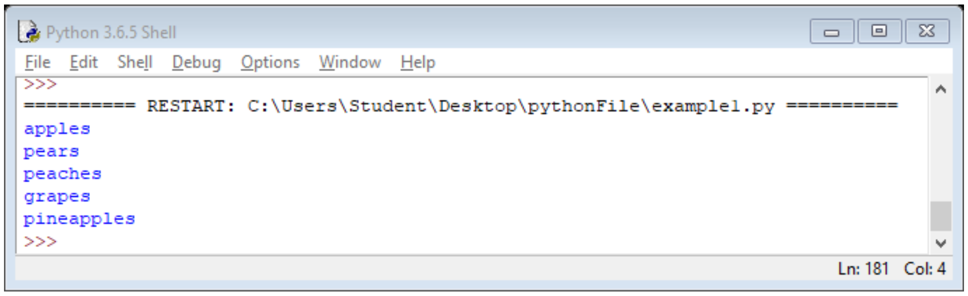

for loop declaration
range set the initial value for x to be 0 and increasing repeat the loop block code up to 5 (exclusive), including
loop block
x is the iteration variable or counter variable

```
for x in range(0,5):
    print('Hello ', x)
```

We can also use the *for* loop to print lists. For example, we can create a list of *fruits* and use *for* loop to print each of the value inside the list:

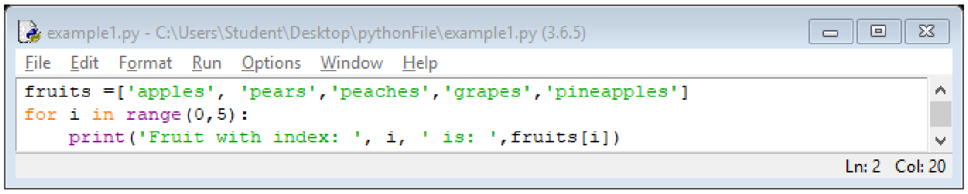

example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 16

```
fruits =['apples', 'pears','peaches','grapes','pineapples']
for i in fruits:
    print(i)
```

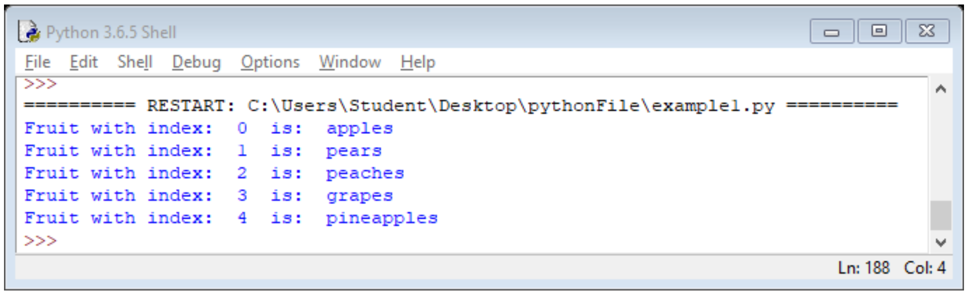


```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
apples
pears
peaches
grapes
pineapples
>>>
```

We can also print the list as the following:


example1.py - C:\Users\Student\Desktop\pythonFile\example1.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 2 Col: 20

```
fruits =['apples', 'pears','peaches','grapes','pineapples']
for i in range(0,5):
    print('Fruit with index: ', i, ' is: ',fruits[i])
```



```
>>>
========== RESTART: C:\Users\Student\Desktop\pythonFile\example1.py ==========
Fruit with index:  0  is:  apples
Fruit with index:  1  is:  pears
Fruit with index:  2  is:  peaches
Fruit with index:  3  is:  grapes
Fruit with index:  4  is:  pineapples
>>>
```

The `range()` function with three argument

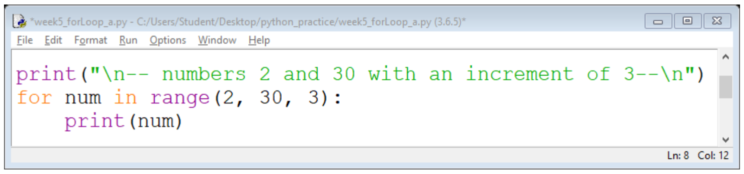
The **`range()`** function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter:

Initial value **Increment**

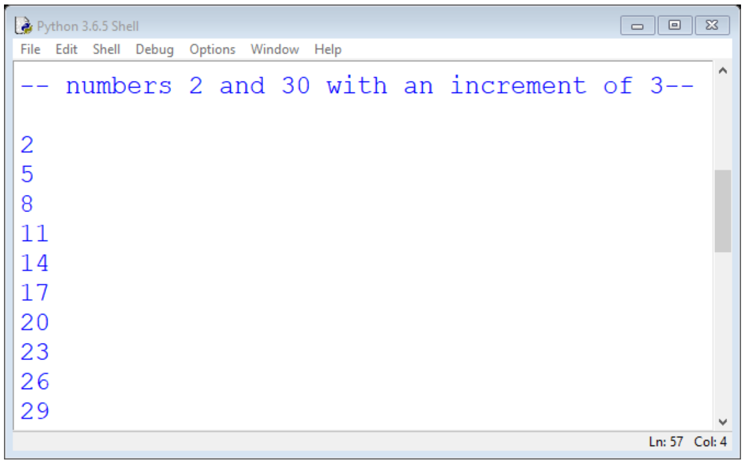
```
for x in range(2, 30, 3):
```

End value (not included)
exclusive value

Example) Use a for loop to print numbers between 2 and 30 with an increment of 3

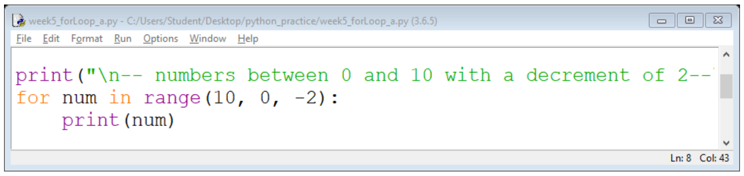

week5_forLoop_a.py - C:/Users/Student/Desktop/python_practice/week5_forLoop_a.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 8 Col: 12

```
print("\n-- numbers 2 and 30 with an increment of 3--\n")
for num in range(2, 30, 3):
    print(num)
```

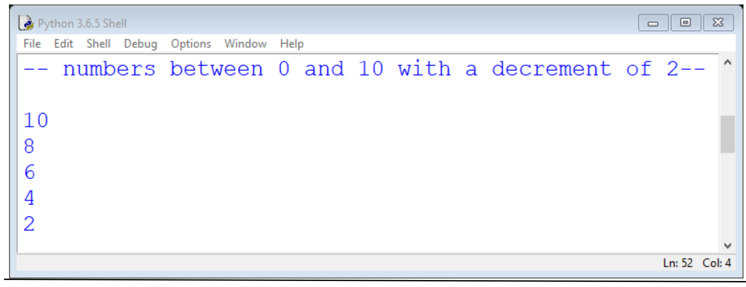


```
-- numbers 2 and 30 with an increment of 3--

2
5
8
11
14
17
20
23
26
29
```

Example) Use a for loop to print numbers between 0 and 10 with a decrement of 2


week5_forLoop_a.py - C:/Users/Student/Desktop/python_practice/week5_forLoop_a.py (3.6.5)
File Edit Format Run Options Window Help
Ln: 8 Col: 43

```
print("\n-- numbers between 0 and 10 with a decrement of 2--
for num in range(10, 0, -2):
    print(num)
```



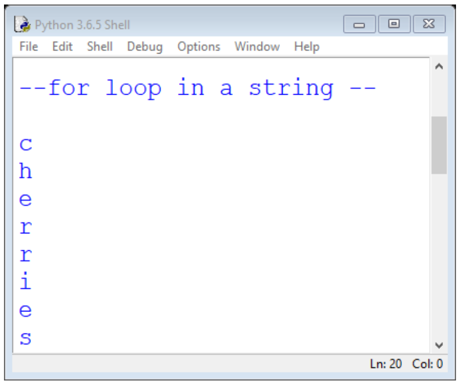
```
-- numbers between 0 and 10 with a decrement of 2--

10
8
6
4
2
```

Looping through a string

Since string is a list of characters, we can use for loop to print of a sequence of characters of a given string.

Example) Use for loop to print each character of **cherries**

```
print("\n--for loop in a string --\n")

for x in "cherries":
      print(x)
```



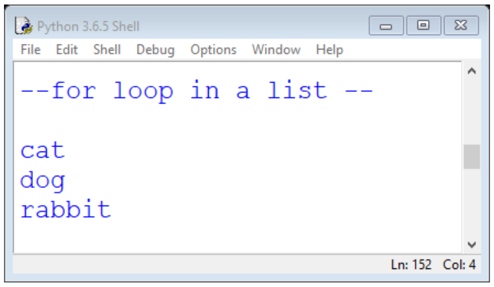
```
--for loop in a string --

c
h
e
r
r
i
e
s
```

Looping through a list

You can loop through the list items by using a for loop from the beginning to the end of the list without using **`range()`** function.

Example) Print each item in list **`animals`**

```
print("\n--for loop in a list --\n")

animals=["cat","dog","rabbit"]

for counter in animals:
    print(counter)
```



```
--for loop in a list --

cat
dog
rabbit
```

The break and continue statement

The other way to stop a loop is using a **`break`**. A *break* will terminate a loop if the condition meets the break even if the while condition is true.

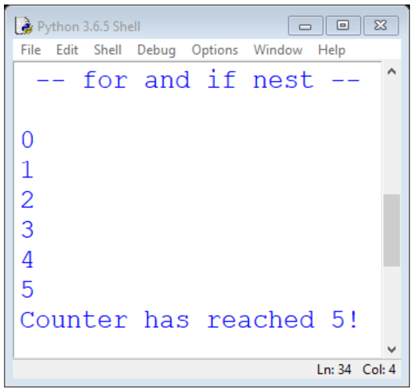
The **`continue`** statement we can stop the current iteration, and continue with the next iteration

Nesting for loop and if statement

We can also nest an `if` statement inside of a `for` loop.

Example) Use a for loop to print numbers from 0 to 10 exclusive and break the for loop when the counter reaches to 5

```
print("\n -- for and if nest --\n")

for counter in range(11):
    print(counter)
    if counter == 5:
        print("Counter has reached 5!")
        break
```

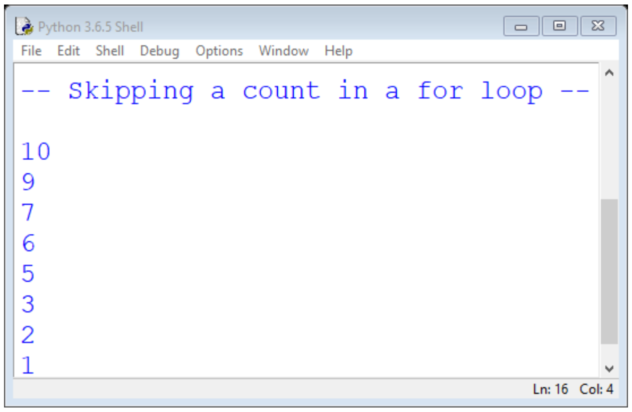


```
 -- for and if nest --

0
1
2
3
4
5
Counter has reached 5!
```

`break` in the **`if`** statement ends the program when the **`if`** condition is TRUE.

Example) Use a `for` loop to print numbers from 10 to 0 and skip numbers that are multiple of 4.

```
print("\n-- Skipping a count in a for loop --\n")

for counter in range(10,0,-1):
    if counter%4==0:
        continue
    print(counter)
```



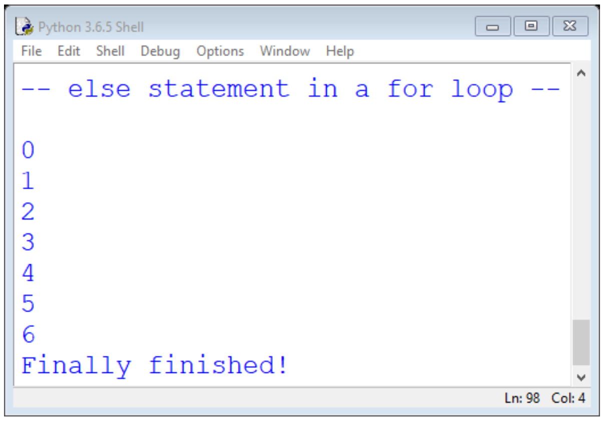
```
-- Skipping a count in a for loop --

10
9
7
6
5
3
2
1
```

else statement in a for loop

The **`else`** keyword in a **`for`** loop specifies a block of code to be executed when the loop is finished.

Example) Print all numbers from 0 to 6, and print a message **`Finally finished!`** when the loop has ended

```
print("\n-- else statement in a for loop --\n")

for x in range(7):
    print(x)
else:
    print("Finally finished!")
```



```
-- else statement in a for loop --

0
1
2
3
4
5
6
Finally finished!
```

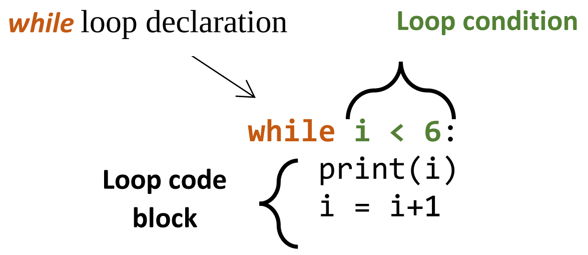
WHILE LOOP

Python has two primitive loop commands: for loop and while loop. A *for* loop is a loop of a specific length, whereas a *while* loop is a loop that is used when we do not need the loop anymore.

When a program is in a loop, it performs an operation repeatedly as long as a condition is true. A while loop evaluates the condition and executes the statement if that condition is true. Then it repeats that operation until condition evaluates to false.

The syntax of the *while* loop is as the following:

1. Check the condition is true
2. Execute the code in the while loop block
3. Set the condition again.
4. Repeat from step 1.



```
while i < 6:
    print(i)
    i = i+1
```

Counting in while loop

To repeat a task in a specific number of times using a while loop:

1. We will have to setup a counter before the loop
2. Check to see if the counter before and each time we loop.
3. Update the counter each time we loop.

Parts of the while loop

For every while loop to function properly three component must exist:

1. Initialization
2. Comparison: done repeatedly to check for termination condition
3. Update: changes the condition every time, makes loop meaningful

Python Assignment Operators

The Python assignment operators are used to assign values to the declared variables

Python Assignment Operators				
Symbol	**Operations**	**Python Example**	**Arithmetic equality**	**Value**
`=`	Assign a value	`x = 5`	`x = 5`	5
`+=`	Self-addition	`x+=3`	`x = x+3`	8
`-=`	Self-subtraction	`x-=3`	`x = x-3`	2
`*=`	Self-multiplication	`x*=3`	`x = x*3`	15
`/=`	Self-division	`x/=3`	`x = x/3`	$1.6\overline{6}$
`//=`	Self-division, returning only the quotient	`x//=3`	`x = x//3`	1
`%=`	Returning remainder of a self-division	`x%=3`	`x = x%3`	2
`**=`	Self-exponential	`x**=3`	`x = x**3`	125

Example) create a while loop that will run as long as the counter ***i*** is less than 6:

```
i=0
while i<6:
    print(i)
    i = i+1  # i +=1
```

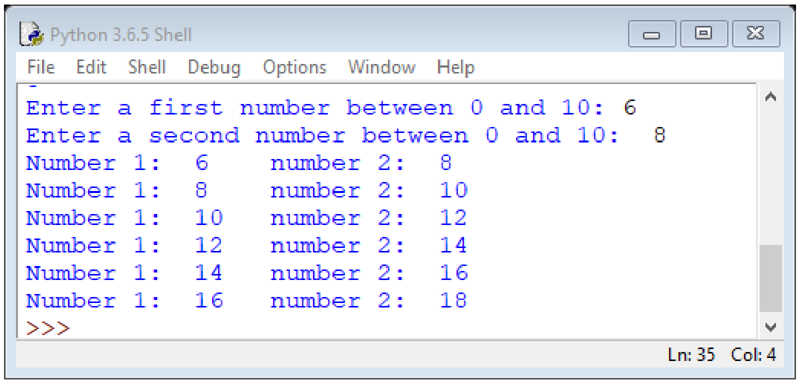
Result

```
0
1
2
3
4
5
>>>
```

Example) Create a python code that will ask the user to Enter two numbers between 0 and 10 and use a while loop to increment each number by 2 until both of them reach up to 25.

```
number1 = int(input("Enter a first number between 0 and 10: "))
number2 = int(input("Enter a second number between 0 and 10:  "))
while number1<20 and number2<20:
    print("Number 1: ", number1, "\t number 2: ", number2)
    number1 += 2
    number2 += 2
```

Result

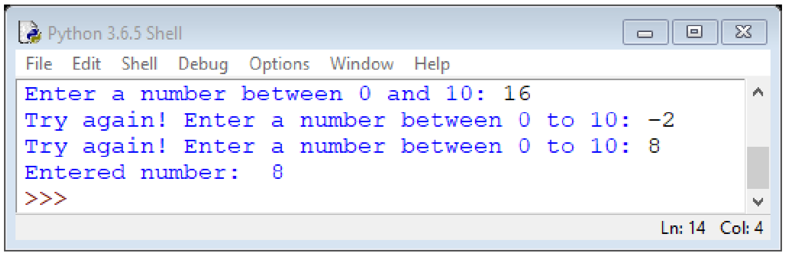


```
Enter a first number between 0 and 10: 6
Enter a second number between 0 and 10:  8
Number 1:  6    number 2:  8
Number 1:  8    number 2:  10
Number 1:  10   number 2:  12
Number 1:  12   number 2:  14
Number 1:  14   number 2:  16
Number 1:  16   number 2:  18
>>>
```

Example) Create a Python code using while loop that will check if a number is within 0 and 10

```
number = int(input("Enter a number between 0 and 10: "))
while number<0 or number>10:
    number = int(input("Try again! Enter a number between 0 to 10: "))

print("Entered number: ", number)
```

Result

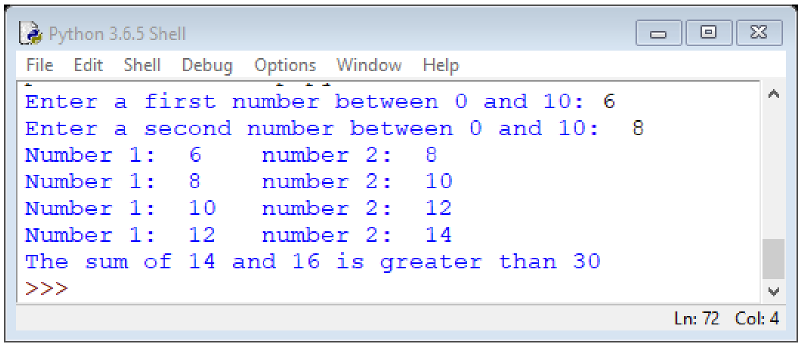


```
Enter a number between 0 and 10: 16
Try again! Enter a number between 0 to 10: -2
Try again! Enter a number between 0 to 10: 8
Entered number:  8
>>>
```

Example) We can have a *while* program from the previous example and the program will stop if the sum of the first and second number is greater than or equal to 30. To run this program, we can have an *if* statement inside the while loop:

```
number1 = int(input("Enter a first number between 0 and 10: "))
number2 = int(input("Enter a second number between 0 and 10:  "))
while number1<20 and number2<20:
    print("Number 1: ", number1, "\t number 2: ", number2)
    number1 += 2
    number2 += 2
    if number1 + number2 >=30:
        print("The sum of %s and %s is greater than 30" %(number1,number2))
        break
```

Result

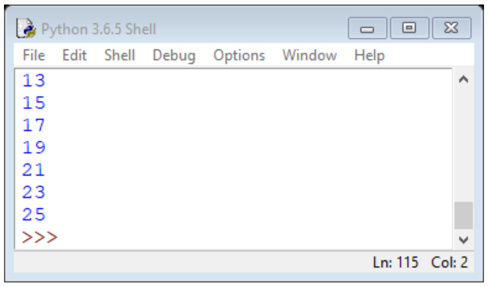


```
Enter a first number between 0 and 10: 6
Enter a second number between 0 and 10:  8
Number 1:  6     number 2:  8
Number 1:  8     number 2:  10
Number 1:  10    number 2:  12
Number 1:  12    number 2:  14
The sum of 14 and 16 is greater than 30
>>>
```

Example) Write a Python program using a while to print all odd numbers between 12 and 25

```
x = 11
while x<=25:
    x += 1
    if x%2==0:
        continue
    print(x)
```

Result



```
13
15
17
19
21
23
25
>>>
```

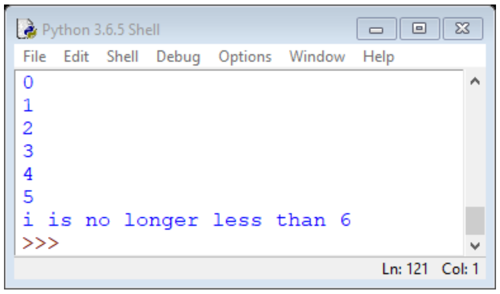
<u>The else statement</u>

With the else statement we can run a block of code once when the condition no longer is true.

Example) Use while loop to print all number less than 6 starting from 0 with an increment of 1.

```
i = 0
while i < 6:
    print(i)
    i += 1
else:
    print("i is no longer less than 6")
```

<u>Result</u>



```
0
1
2
3
4
5
i is no longer less than 6
>>>
```