

Getting Started with Python and IDLE



Some thoughts about programming

"The only way to learn a new programming language is by writing programs in it."

-- B. Kernighan and D. Ritchie

"Computers are good at following instructions, but not at reading your mind."

-- D. Knuth

Program:

- *n.* A magic spell cast over a computer allowing it to turn one's input into error messages.
- *tr. v.* To engage in a pastime similar to banging one's head against a wall, but with fewer opportunities for reward.

What is Python?

- A high-level language developed by Guido van Rossum in the Netherlands in the late 1980s. Released in 1991.
- Named after Monty Python
- Twice received recognition as the language with the largest growth in popularity for the year (2007, 2010)
- Clean, concise syntax
 - Easy to learn
 - Can create powerful programs quickly
- Design is compact
 - Language constructs are easy to learn
 - Easy to remember

Getting Python

If you plan on using one of the Linux systems on campus, you can create a program (such as `“myprog.py”`) using your favorite editor (EMACS, vi, etc.) and run it using the command `"python myprog.py"`.

If you want to install it on your personal computer / laptop, you can download it for **free** at:

www.python.org/downloads

- It's available for both Windows and Mac OS.
- If you have a Mac, you may already have it preinstalled.
- It comes with an editor and user interface called IDLE.

Interpreted vs. Compiled languages

High-level computer languages can be placed into two categories:

- *Interpreted* languages:
 - Each command is translated to machine language individually and executed "on the fly"
- *Compiled* languages:
 - All commands are translated into machine language first
 - The resulting program is stored in a new file
 - Then the machine language program is executed

Python is an interpreted language.

- It uses an *interpreter*, which is a program that translates each command in your program into machine language and executes it.

How Python Works

Consequently, there are actually two different ways to execute Python code:

- Use Python interactively by entering one command at a time.
- Type your source code in a file (that ends in extension .py) and invoke the python interpreter to translate your instructions into machine language and execute them.

Using IDLE:

When you start up IDLE, you will get a window called “**Shell**”. The Shell window is where you will interact with your program.

“**File → New File**”: gets you another window called “**Untitled**”.

- This is where you enter and edit multiple Python instructions to write a complete program.

“**Run -> Run Module**”

- Any output from your program will appear in the Shell window.
- If you are expected to provide your program with input, it will expect you to type it into the Shell window.

“**File -> Save As...**”

- Give your file a name ending in “.py”
- I find it easiest to save the file to “Desktop” to make it easy to find, and then drag-and-drop it somewhere else later

Basics of the Python Language



The framework of a simple Python program

```
def main():
```

This tells the interpreter that you're going to start defining your main program.

```
    Python statement  
    Python statement  
    Python statement  
    Python statement  
    Python statement  
    Python statement
```

These are the instructions that make up your program. You **indent all of the statements by the same amount** to show Python where the list begins and where it ends.

```
main()
```

After you've defined your program, this instruction means, "Now execute it."

Producing output: `print()`

The `print` statement is used to produce output. The basic format is:

```
print (<list of strings, numbers, and  
expressions separated by commas>)
```

The parentheses are part of the statement, the parts in *<angle brackets and italics>* are not.

```
def main():  
  
    print ("Hello world")  
    print ("Hello", "world")  
    print ('The year is: ', 2016, "A.D.")  
    print (72*8-6)  
  
main()
```

Strings

Any sequence of characters between matching quotes is called a `str` for “string”.

You can use either double quotes (“) or single quotes (‘) to mark the start and end. So the two statements:

```
"Hello world"  
'Hello world'
```

represent the same thing: you can use either type of quote marks, as long as they match each other.

Note that the quote marks are not printed. They’re just markers to indicate the start and the end of the `str`.

Special characters

`\n` is a symbol for the ASCII character “line feed” <LF>.

The backslash is used to indicate an “escape sequence”, identifying the next character as something special.

- `\n`: the “new line” escape sequence (like hitting return)
- `\t`: the tab escape sequence (like hitting tab)
- `\"`: double quote (print a real double quote instead of beginning or ending a string)
- `\'`: single quote (print a real apostrophe instead of beginning or ending a string)
- `\\`: a real backslash, if you ever have the need to print one

Arithmetic expressions

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Examples:

```
print (1234 * 56)
```

```
print (87.3 / 7.4)
```

```
print (87 / 7)
```

Arithmetic expressions (cont.)

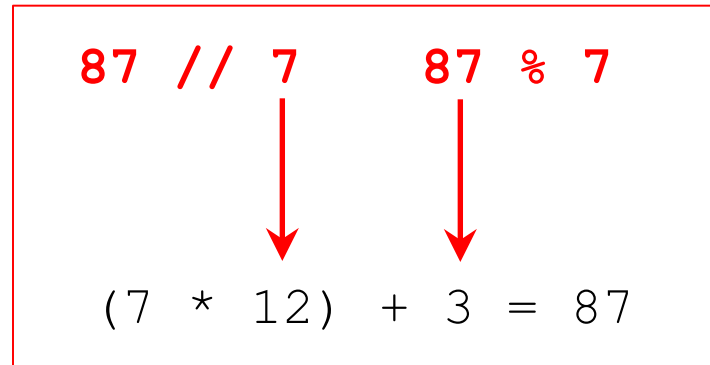
```
print (87 // 7)      = 12
```

```
print (87 % 7)       = 3
```

Can you guess what the operators `//` and `%` do?

`//` Integer division

`%` Remainder



Order of Precedence

The order that operators are calculated is the same as in math:

()	done first
**	↓
*, /, //, %	
+, -	done last

“Ties” are evaluated from left to right.

Variables: What can we name them?

A *variable* is a symbol that denotes a place in memory where a value is stored.

- Variable names must start with a letter or the underscore (“_”) character.
- After that, it can be followed by any number of letters, underscores, or digits.
- Variable names are case-sensitive, so “score” is different from “Score”.
- You must avoid using reserved words as variable names: these are words that have a special meaning in a programming language such as Python.
 - For example: `def`, `str`, `print`, etc.
 - IDLE displays reserved words in color to help you recognize them, which is useful since most people don’t know all of them.

Variables: Naming Conventions

In addition to the hard-and-fast rules on the previous chart, there are also naming conventions that all (good) programmers obey:

- You should choose meaningful names that describe what the purpose of the variable is. This helps people reading the program (including you) understand what the code is doing.
 - Use `max` rather than `m`
 - Use `item` rather than `c`
 - Some exceptions
- Variable names should begin with a lowercase letter.
- It is common to combine multiple words (such as `avgHeight`) into a variable name in order to be descriptive. When you do this, improve readability by using lowercase for the first “word” and uppercase for subsequent words. (This is called “camelCase”.)

Data Types



What is a data type?

A *data type* is the type of value represented by a constant or stored by a variable.

So far, you have seen the following three basic (concrete) data types:

- `str`: represents text (a string)
 - Currently, we use it for input and output
 - You'll see more uses for it later if we have time
- `int`: whole numbers
 - Computations are exact
- `float`: real numbers (numbers with decimal points)
 - Large range, but fixed precision
 - Computations are not exact
 - Example: $1.0 / 3.0 = .3333333333333333$

Data Types and Variables

You do not "declare" variables in Python!

- A variable is simply a name or placeholder for a location or locations in memory.
- The placeholder can hold any type of data you want.
- If you assign a value to a variable that is currently holding a value of a different type, Python won't care. It will accept the new value and its type without question.

Data Types and arithmetic operations

Most arithmetic operations behave as you would expect for all data types.

- Combining two `floats` results in an `float`.
- Combining two `ints` results in an `int` (provided you do division with `//`).
- `float` division is an exception: it behaves as you probably want it to. For instance, `5 / 2` gives you `2.5`.

Python will figure out what the result should be and make the result the appropriate data type.

Data Type Conversion

Sometimes, you want the result to be a different type than the one Python assumes.

For this, Python provides functions to *explicitly* convert numbers from one type to another:

```
float (<number or variable>)
```

```
int (<number or variable>)
```

```
str (<number or variable>)
```

Note: int truncates, meaning it throws away the decimal point and anything that comes after it. If you need to *round off* to the nearest whole number, use:

```
round (<number or variable>)
```

Data Types: Examples of Explicit Conversion

What is the output?

```
float(3)
```

```
int(3.9)
```

```
round(3.9)
```

```
int(45.2)
```

```
round(45.2)
```

```
str(17)
```

```
int("1212")
```

```
int("Hello world")
```

Question: Data Types

What is the output of the following code?

```
result = 5.0 + 11//2  
print (result)
```

Lists

One of the most important and versatile data types in Python is the *list*.

It's essentially a bunch of items, separated by commas, and placed between square brackets.

```
list1 = [ "a", "b", "c", "d", "e" ]  
list2 = [ 1, 2, 3, 4 ]  
list3 = ["physics", 1992, "chemistry", 5.2]
```

Note that the elements of a list do not have to be the same type.

Accessing Lists

To access values in lists, use the square brackets along with an index or range of indices.

```
list1 = [ "a", "b", "c", "d", "e" ]  
print ("item = ",list1[2])
```

Output:

```
item = c
```

```
list1[3] = "z"  
print ("list1 = ",list1)
```

Output:

```
list1 = ["a","b","c","z","e"]
```

Other Basic List Operations

Python Expression	Result	Description
<code>len([1,2,3])</code>	3	Length
<code>[1,2,3] + [4,5,6]</code>	<code>[1,2,3,4,5,6]</code>	Concatenation
<code>["yo"] * 3</code>	<code>["yo", "yo", "yo"]</code>	Repetition
<code>3 in [1,2,3]</code>	True	Membership

List slicing

```
myList = [ "A", "B", "C", "D" ]
```

Indices: 0 1 2 3 (above) and -3 -2 -1 (below)

Python Expression	Result	Description
<code>myList[2]</code>	<code>C</code>	index starts from zero
<code>myList[1:3]</code>	<code>["B", "C"]</code>	count from the right
<code>myList[1:]</code>	<code>["B", "C", "D"]</code>	from element 1 to the end
<code>myList[:2]</code>	<code>["A", "B"]</code>	from the beginning to element 2
<code>myList[-3:]</code>	<code>["B", "C", "D"]</code>	third element from the right to the end

Keyboard Input



Keyboard Input

The `input()` function is used to read data from the user during program execution.

Format:

```
input (<prompt string>)
```

When it's called:

- It displays the “prompt string”, a `str`. The intent is that it should be a message to the user that the program is waiting for the user to type in a string.
- It will wait until the user types something and hits the “Enter” or “Return” key.
- It returns whatever the user typed as a `str` as a *return value*.

Example of `input()`

```
def main():  
    userName = input ("Please enter your name: ")  
    print (userName)  
  
main()
```

This will:

- Display the string "Please enter your name: "
- Wait until the user types something (ending with the "Enter" key)
- Stores whatever the user types in the variable `userName`.
- Prints the value of `userName`.

More on `input()`

```
extraCredit = 10
examScore = input("Please enter exam score: ")
print ("Final score = ", examScore + extraCredit)
```

This will result in an error message. Since `input()` returns a `str`, you can't do arithmetic with it!

Note that assigning a `str` to `examScore` is not an error. Variables are not typed. This only becomes an error when we try to add a `str` to an `int`.

In order to compute the sum, first convert the `str` into an `int` by using the `int` conversion function:

```
extraCredit = 10
examScore = int(input("Please enter exam score: "))
print ("Final score = ", examScore + extraCredit)
```


Comments and Continuation Characters



A simple program to illustrate comments:

```
# Example 1
# CS313E
# Spring 2016

def main():

    # ask user for three items
    g1 = int(input("What is the cost of the first item? "))
    g2 = int(input("What is the cost of the second item? "))
    g3 = int(input("What is the cost of the third item? "))

    # calculate the total
    sumOfCosts = g1 + g2 + g3

    # calculate tax
    tax = .075 * sumOfCosts          # assumes tax rate is 7.5%

    # print out results
    print ("Sum of items is: ", sumOfCosts)
    print ("Tax of items is: ", tax)
    print ("Grand total is: ", sumOfCosts+tax)

main()
```

Line Continuation

If your line of Python code is too long, you can extend it using the backslash character “\”

- Place it at the end of the line and it “escapes” the carriage return at the end.
- The Python interpreter will assume the next line is part of the same line.

Example:

```
Name = "Jarvis"
print ("Hello, my name is ", name, \
      " How are you?")
```

produces:

```
Hello, my name is Jarvis.  How are you?
```

Decision Structures



Comparisons

There is another type in Python (in addition to `int`, `float`, and `str`) called `bool`.

- `bool` is short for “Boolean”, which is kind of logical algebra invented by George Boole.
- There are two possible values for a Boolean constant or variable: `True` and `False`.

Boolean expressions come into play when we compare two values. You can compare:

- Numbers or Strings
- Literals or variables
- Expressions

How do you specify a comparison?

You compare two things using a *relational operator*:

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equality
!=	Not equals

Examples of comparisons

`18 < 15`

`False`

`101 >= 99`

`True`

`(4 == 2+2)`

`True`

`(15 != 3*5)`

`False`

`12.0 == 3.0 * 4.0`

`True (we hope)`

`"a" < "b"`

`True`

ASCII table

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	_	127	7F	DEL

Lexicographic Order

- Strings are rated according to *lexicographic* order, not dictionary order
- Words are ordered from A-Za-z
 - Capital letters first in alphabetical order
 - Lower-case letters second in alphabetical order
 - This means all upper-case letters come before all lower-case letters

The Python conditional statement: `if`

```
def main():
```

```
    command
```

```
    command
```

```
    if <condition> :
```

```
        command
```

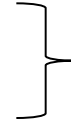
```
        command
```

```
        command
```

```
    command
```

```
    command
```

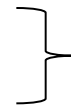
```
main()
```



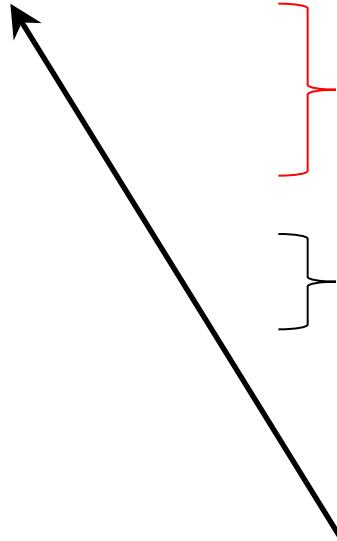
Statements not dependent
on the condition



Statements only executed
if the condition is true



Statements not dependent
on the condition



note the colon (":")

Indentation is very important!

An example of an `if` statement:

```
number = int(input("Input a number: "))  
if number > 10:  
    print (number, "is greater than 10!")  
print ("done")
```

Output if we type in 25?

```
Input a number: 25  
25 is greater than 10!  
done
```

Output if we type in 8?

```
Input a number: 8  
done
```

The if-else statement

```
if <condition> :
```

```
    command
```

```
    command
```

```
    command
```

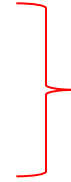
```
else :
```

```
    command
```

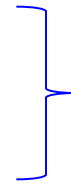
```
    command
```

```
command
```

```
command
```



Statements only executed
if the condition is true



Statements only executed
if the condition is false



Statements not dependent
on the condition

- Note the two colons (":")
- Note the indentation

if-else exercise

Write a complete program that asks the user to enter a number.

- If the number is even, print the number, followed by “ is even”.
- If the number is odd, print the number, followed by “ is odd”.

Hint: use the remainder operator “%” !

```
def main():  
  
    userNumber = int(input("Input a number: "))  
  
    if ((userNumber % 2) == 1):  
        print (userNumber, " is odd")  
    else:  
        print (userNumber, " is even")  
  
main()
```

The if-elif-else statement

```
if <condition> :  
    command  
    command  
    command
```

```
elif <condition> :  
    command  
    command
```

```
elif <condition> :  
    command  
    command
```

```
else :  
    command  
    command
```

```
command  
command
```



You can have as many of these blocks as you like



These statements are only executed if all of the conditions fail

Example of the if-elif-else statement

```
grade = int(input("Enter your test score:" ))

if grade >= 90 :
    letterGrade = "A"
elif grade >= 80 :
    letterGrade = "B"
elif grade >= 70 :
    letterGrade = "C"
elif grade >= 65 :
    letterGrade = "D"
else :
    letterGrade = "F"

print ("Your grade is: ", letterGrade)
```

Conditionals: nested ifs

You can put `if` statements inside the body of the `if` (or `elif` or `else`) statement:

```
if (<condition>) :
```

```
    if (<some other condition>) :
```

```
        command
```

```
    else:
```

```
        command
```

**Indentation is
very important!**

```
elif (<condition>) :
```

```
...
```

Gotchas with conditionals

- Exactly one of the clauses of an `if-elif-else` statement will be executed
 - Only the first `True` condition
 - Think carefully about the construction of your `if` statements before coding
 - Think about a flowchart: you will only follow ONE arrow at a time
- Be very careful when you compare floats since floating-point arithmetic is not exact.

```
if (.3 == .1+.2) is False
```

What is the expected output?

```
if (125 < 140):  
    print ("first one")  
elif (156 >= 140):  
    print ("second one")  
else:  
    print ("third one")
```

A. first one

C. third one

B. second one

D. first one
second one

Iterative Structures



The `for` statement

The general form of a `for` statement is:

```
for <var> in <some kind of series>:
```

The easiest way to explain this is with an example:

```
for i in [1, 2, 3]:  
    print(i)
```

Produces the output:

1
2
3

- The thing in `[]` is a *list*
- The number of times you go through the loop = the number of items in the list
- Each time you go through the loop, you assign the value of the next item to the variable in the `for` statement
- Don't forget the colon
- Indentation is important!

The `for` statement (cont.)

The list doesn't have to consist of numbers:

```
for name in ["Groucho", "Harpo", "Chico"]:  
    print(name)
```

produces the output:

```
Groucho  
Harpo  
Chico
```

The range function

`range(<number>)` produces a list of `ints` counting from zero up to `<number>-1`.

For example, `range(5)` gives you the list 0, 1, 2, 3, 4.

Example:

```
for number in range(3):  
    print(number)
```

produces the output:

```
0  
1  
2
```

The `range` function: more features

`range(<num1, num2>)` produces a list of `ints` counting from `<num1>` up to `<num2>-1`.

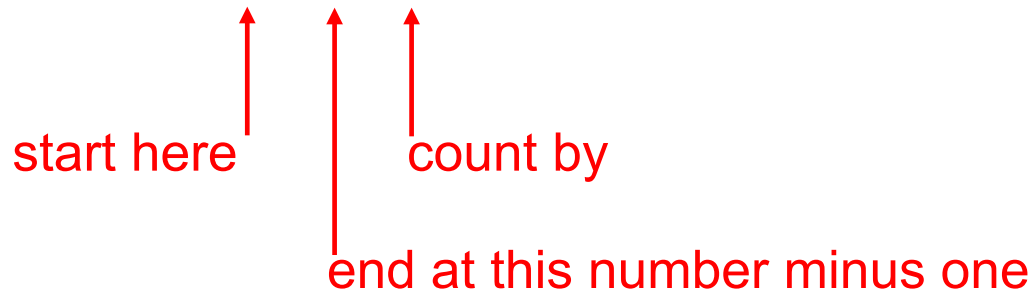
For example, `range(3, 7)` gives you the list 3, 4, 5, 6.



start here end at this number minus one

Finally, `range(<num1, num2, num3>)` produces a list of `ints` counting from `<num1>` up to `<num2>-1` counting by `num3`'s.

For example, `range(4, 36, 5)` gives you the list 4, 9, 14, 19, 24, 29, 34.



start here count by
end at this number minus one

Quick check:

What is the output?

```
def main():  
    for i in [2, "three", 4.0]:  
        print(i)  
main()
```

What is the output?

```
def main():  
    for i in range(3, 22, 3):  
        print(i+1)  
main()
```

Colons and indentation

What do the colon and the indentation signify in the `for` statement?

The colon indicates that the `for` statement should expect a block of statements, not necessarily a single statement, and the start and end of the block will be reflected in the indentation.

This applies to everywhere else we've seen the colon and indentation as well.

```
def main():  
  
    # print out a heading for our table  
    print("# 2    3")  
    print("=====")                # 12 equal signs  
  
    # "range(1,11)" goes from 1 to 10  
    for i in range(1,11):  
        print("-----")          # 12 hyphens  
        print(i, i*i, i**3)  
  
main()
```

Another way to iterate: the `while` statement

The general form of a `while` statement is:

```
while <boolean expression>:
```

Again, the easiest way to explain this is with an example:

```
powerOf2 = 1
while (powerOf2 < 100):
    print(powerOf2)
    powerOf2 = powerOf2 * 2
```

Defining Functions



Defining your own functions

You can define your own functions using the `def` statement.

Example:

```
def printMyName(name):  
    print("*****")  
    print(name)  
    print("*****")
```

If you then invoke (“call”) this function by typing

```
printMyName(" Eric Cartman")
```

it will print

```
*****  
Eric Cartman  
*****
```

Parameters

```
def printMyName(name):  
    print("*****")  
    print(name)  
    print("*****")  
  
def main():  
  
    printMyName("Bill")  
    printMyName("Barack Obama")  
  
    singer = "Justin Bieber"  
    printMyName(singer)  
  
    printMyName(input("Enter a name, please: "))  
  
main()
```


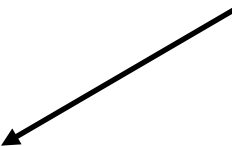
In general:

```
# define all of your functions
def <function>():
def <function>():
def <function>():
def <function>():
    .
    .
    .
# define your main program
def main():
    .
    .
    .
# call your main program to start execution
main()
```

Things to remember about `def`

The `def` statement ends in a colon ("`:`")

```
def printMyName(name) :  
    print("*****")  
    print(name)  
    print("*****")
```



All statements that are part of the function definition must be indented. This is how the Python interpreter knows that these statements are associated with the `def` above it, and not part of some other part of your program.

Return values

You return a value from a function by using the `return` statement.

Example:

```
def sumOfNumbers(num1, num2):  
    total = num1 + num2  
    return total
```


Call this function by typing

```
sum = sumOfNumbers(7.3, 83.2)  
print (sum)
```

90.5

Parameter Passing in Python

The parentheses of a function contain zero or more *parameters* that are passed from the calling statement.



```
def printMyName(name) :  
    print("*****")  
    print(name)  
    print("*****")
```

You have probably learned about two methods of parameter passing in your previous programming class: *pass-by-value* and *pass-by-reference*. Python uses neither: it uses pass-by-object.

We will revisit this later.