

DATA, EXPRESSIONS, STATEMENTS

2

Python interpreter and interactive mode; values and types: int, float, boolean, string, and list; variables, expressions, statements, tuple assignment, precedence of operators, comments; modules and functions, function definition and use, flow of execution, parameters and arguments; Illustrative programs: exchange the values of two variables, circulate the values of n variables, distance between two points.

2.1 Python interpreter and interactive mode

One of the challenges of getting started with Python is that you might have to install Python and related software on your computer. If you are familiar with your operating system, and especially if you are comfortable with the command-line interface, you will have no trouble installing Python. But for beginners, it can be painful to learn about system administration and programming at the same time.

There are two versions of Python, called Python 2 and Python 3. They are very similar, so if you learn one, it is easy to switch to the other. In fact, there are only a few differences you will encounter as a beginner. This book is written for Python 3, but includes some notes about Python 2.

2.1.1 Python interpreter

- The Python *interpreter* is a program that reads and *executes Python code*.
- Depending on your environment, you might start the interpreter by clicking on an icon, or by typing `python` on a command line.
- When it starts, you should see output like this:

```
Python 3.4.0 (default, Jun 19 2017, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- The first three lines contain information about the interpreter and the operating system it's running on.
- Check that the **version number**, which is 3.4.0 in this example, begins with 3, which indicates that you are running Python 3.
- If it begins with 2, you are running Python 2.
- The last line is a **prompt** that indicates that the interpreter is ready for you to enter code.
- If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 1 + 1
2
```

2.1.1.1 The first program

- The first program you write in a new language is called “Hello, World!” because all it does is display the words “Hello, World!”. In Python, it looks like this:

```
>>> print('Hello, World!')
```

- This is an example of a **print statement**, although it doesn't actually print anything on paper. It displays a result on the screen. In this case, the result is the words

```
Hello, World!
```

- The quotation marks in the program mark the beginning and end of the text to be displayed; they don't appear in the result.
- The parentheses indicate that print is a function. In Python 2, the print statement is slightly different; it is not a function, so it doesn't use parentheses.

```
>>> print 'Hello, World!'
```

2.1.2 How to execute python

- **Interactive mode**, which means that you interact directly with the *interpreter*.
- Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.
- The alternative is to save code in a file called a *script* and then run the interpreter in *script mode* to execute the script.
- By convention, Python scripts have names that end with *.py*.
- Python provides both modes.
- But there are differences between interactive mode and script mode that can be confusing.
- For example, you might type

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

- The first line assigns a value to miles, but it has no visible effect.
- The second line is an expression, so the interpreter evaluates it and displays the result.
- It turns out as about 42 kilometers.
- But if you type the same code into a script and run it, you get no output at all.
- In script mode an expression, all by itself, has no visible effect.
- Python actually evaluates the expression, but it doesn't display the value unless you tell it to:

```
miles = 26.2
print(miles * 1.61)
```

- A *script* usually contains a *sequence of statements*.
- If there is more than one statement, the results appear one at a time as the statements execute.

For example, the script

```
print(1)
x = 2
print(x)
```

produces the output

```
1
2
```

- The assignment statement produces no output.
- After executing all the statements the output is displayed.

There are three different ways of working in Python:

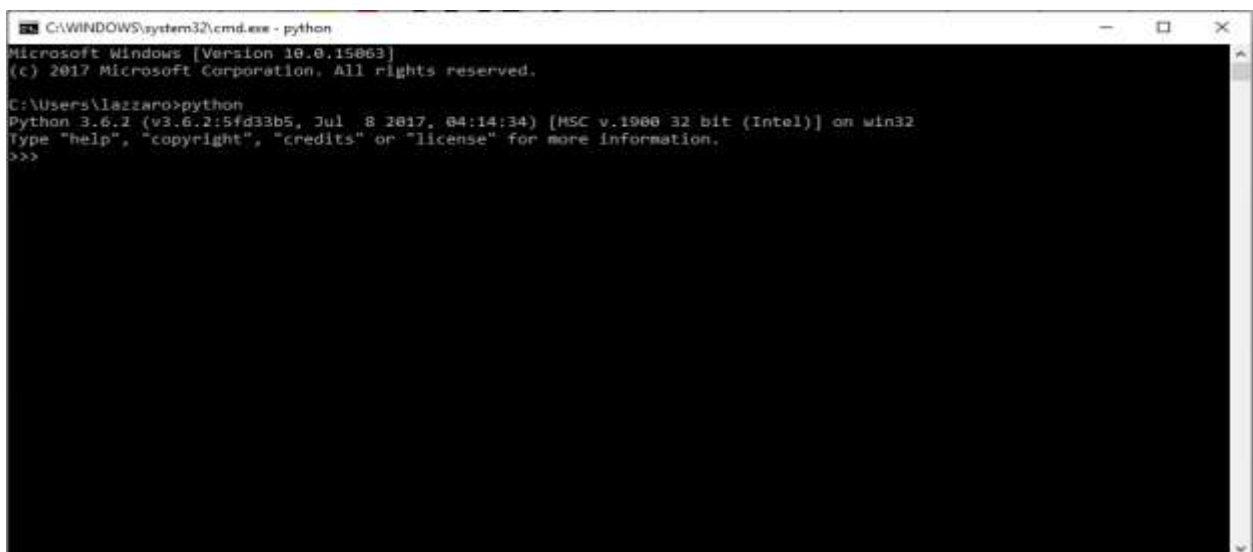
2.1.2.1 Interactive Mode:

You can enter python in the *command prompt* and start working with Python.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.
C:\Users\lazzaro>python
```

- Press Enter key and the Command Prompt will appear like:



```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.
C:\Users\lazzaro>python
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

- Now you can execute your Python commands.

Eg:

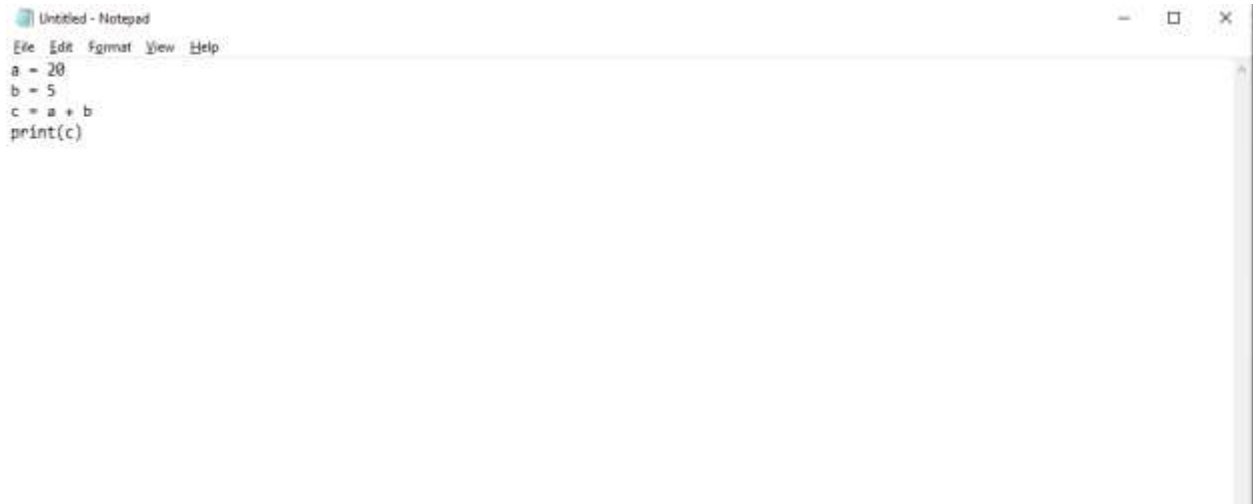


```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\lazzaro>python
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 20
>>> b = 5
>>> c = a + b
>>> print(c)
25
>>> _
```

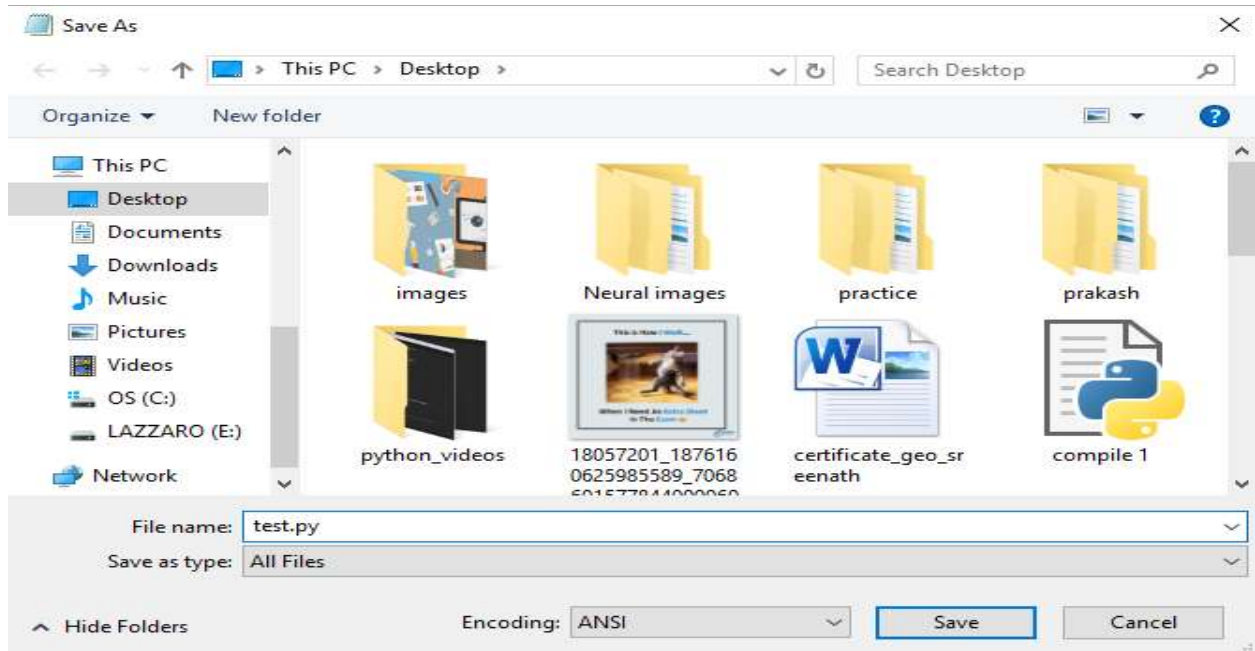
2.1.2.2 Script Mode:

- Using Script Mode , you can write your *Python code in a separate file* using any editor of your Operating System.

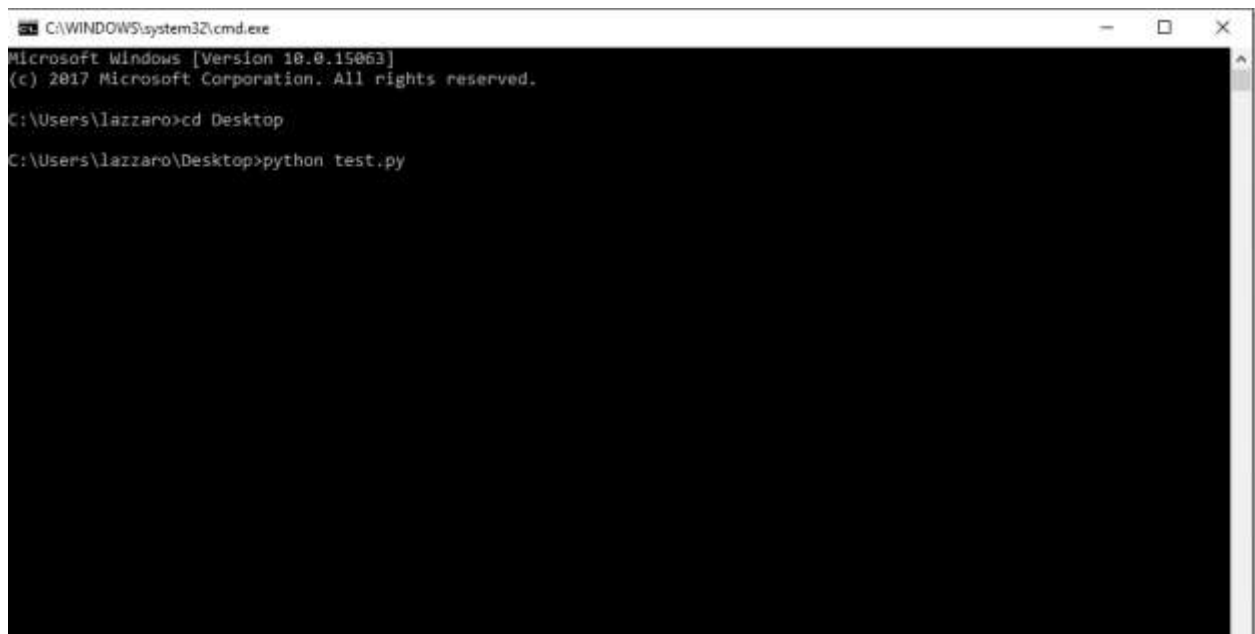


```
Untitled - Notepad
File Edit Format View Help
a = 20
b = 5
c = a + b
print(c)
```

- Save it by *.py extension*.



- Now open Command prompt and execute it by :

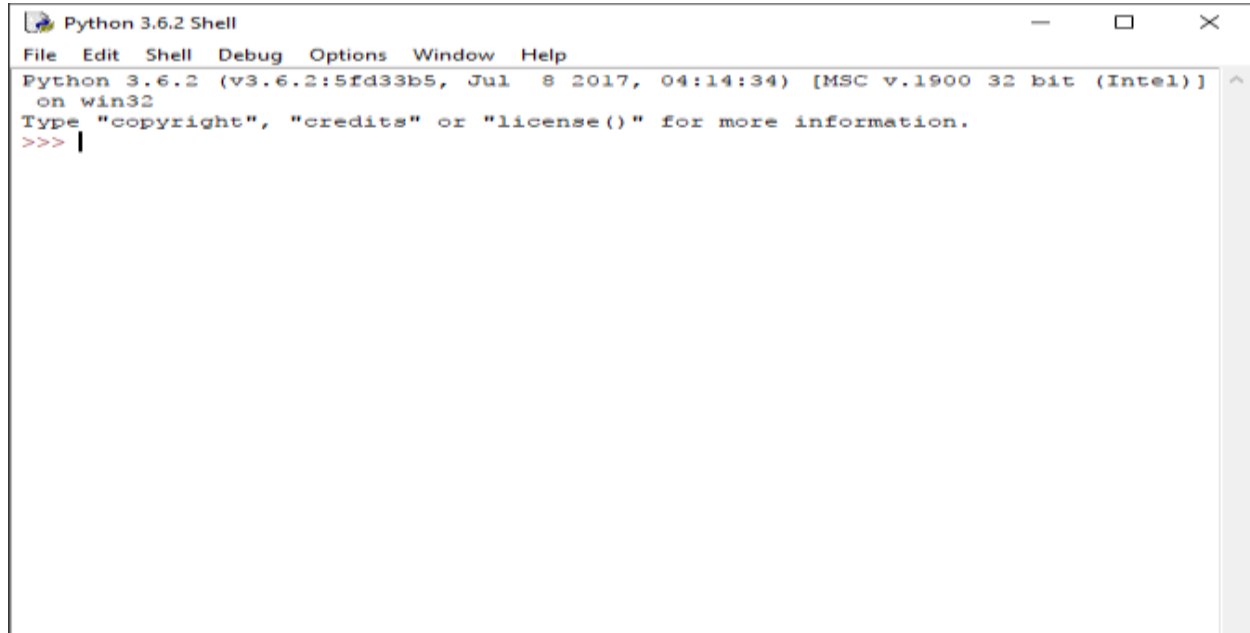


- Path in the command prompt should be where you have saved your file. In the above case file should be saved at desktop.

2.1.2.3 Using IDE: (Integrated Development Environment)

- You can execute your Python code using a **Graphical User Interface (GUI)**.
- All you need to do is:

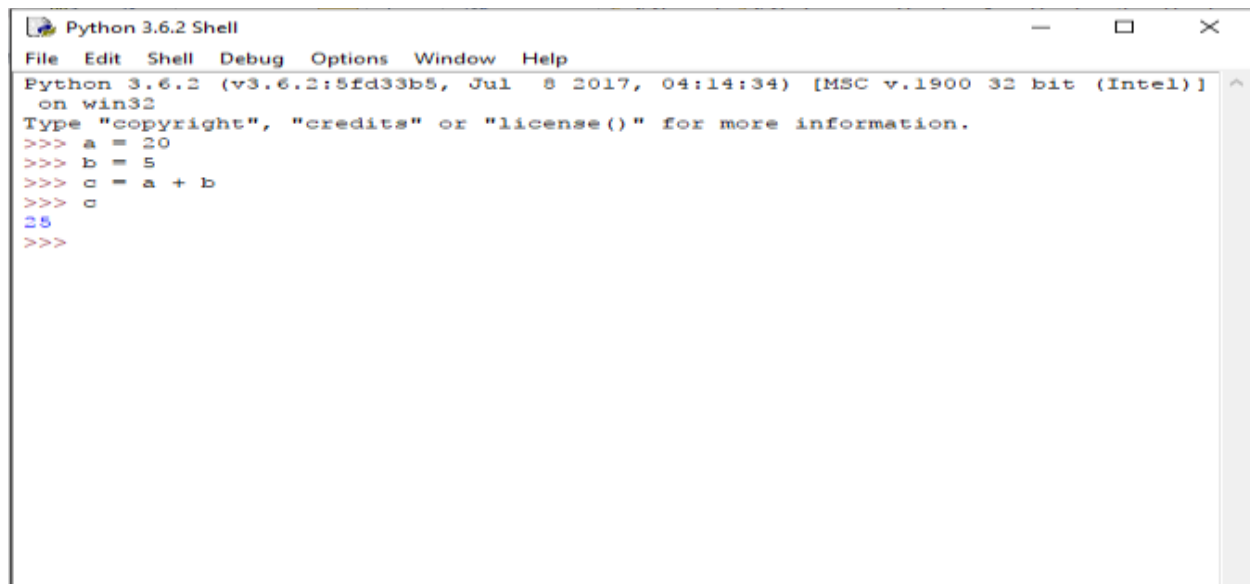
Click on Start button -> All Programs -> Python -> ***IDLE(Python GUI)***



➤ You can use both Interactive as well as Script mode in IDE.

1) Using Interactive mode:

Execute your Python code on the Python prompt and it will display result simultaneously.

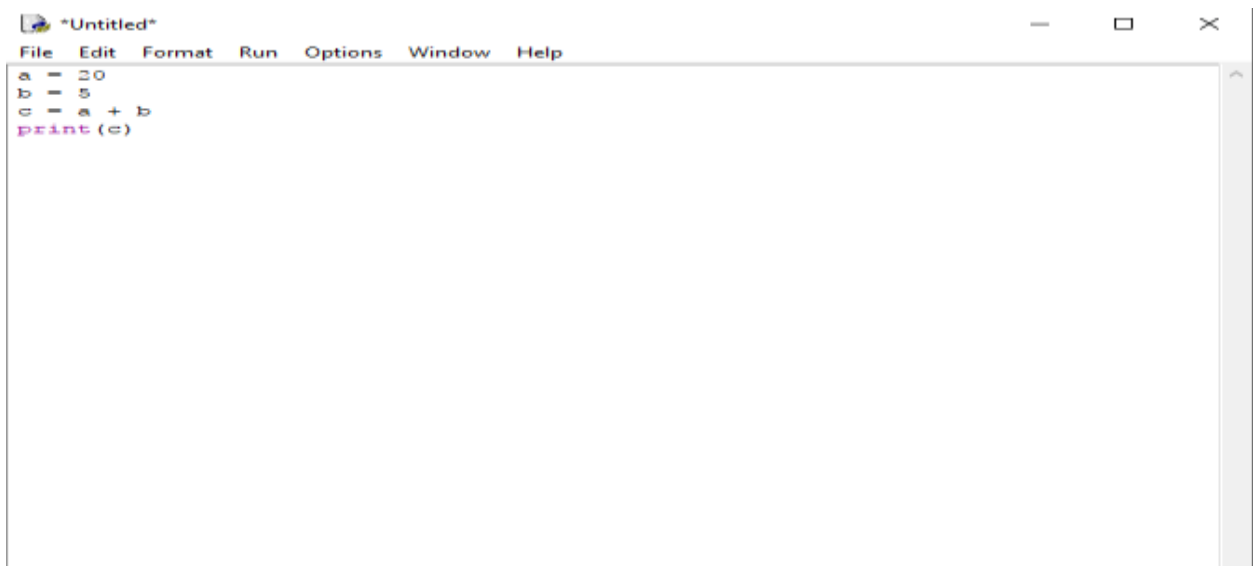


2) Using Script Mode:

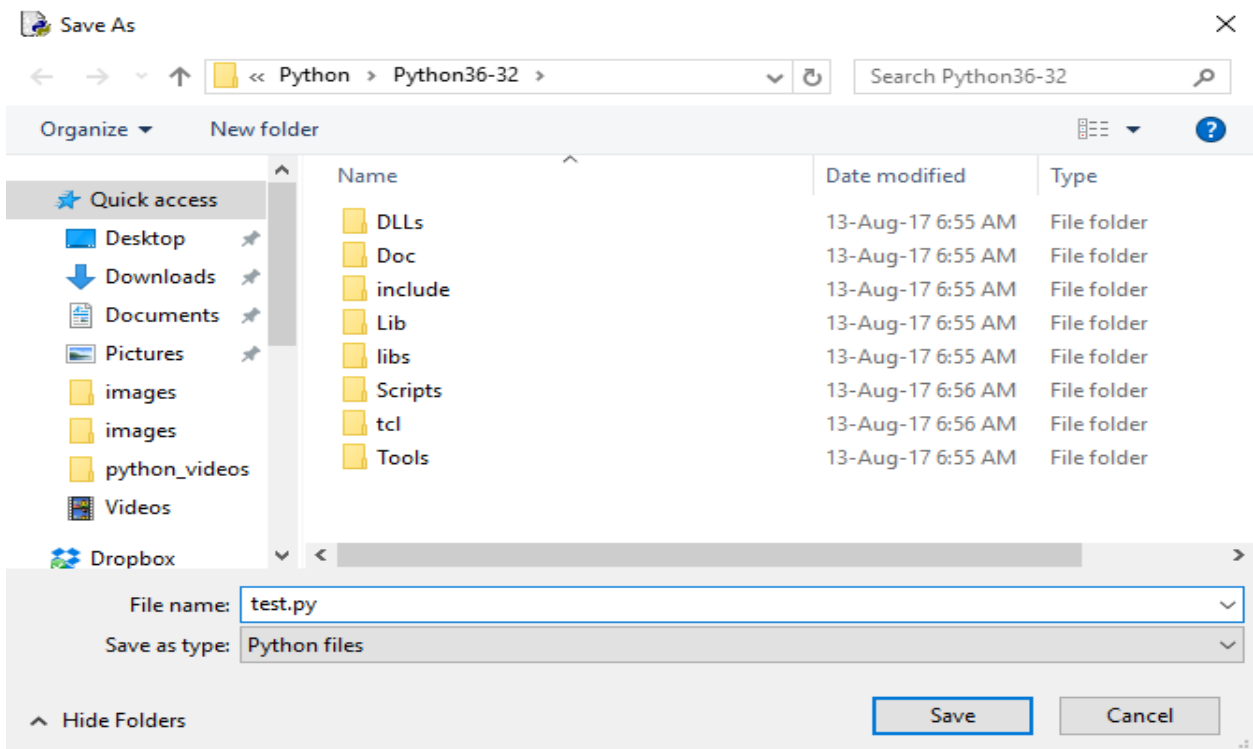
i) Click on Start button -> All Programs -> Python -> IDLE(Python GUI)

ii) Python Shell will be opened. Now click on File -> New Window.

A new Editor will be opened . Write your Python code here.

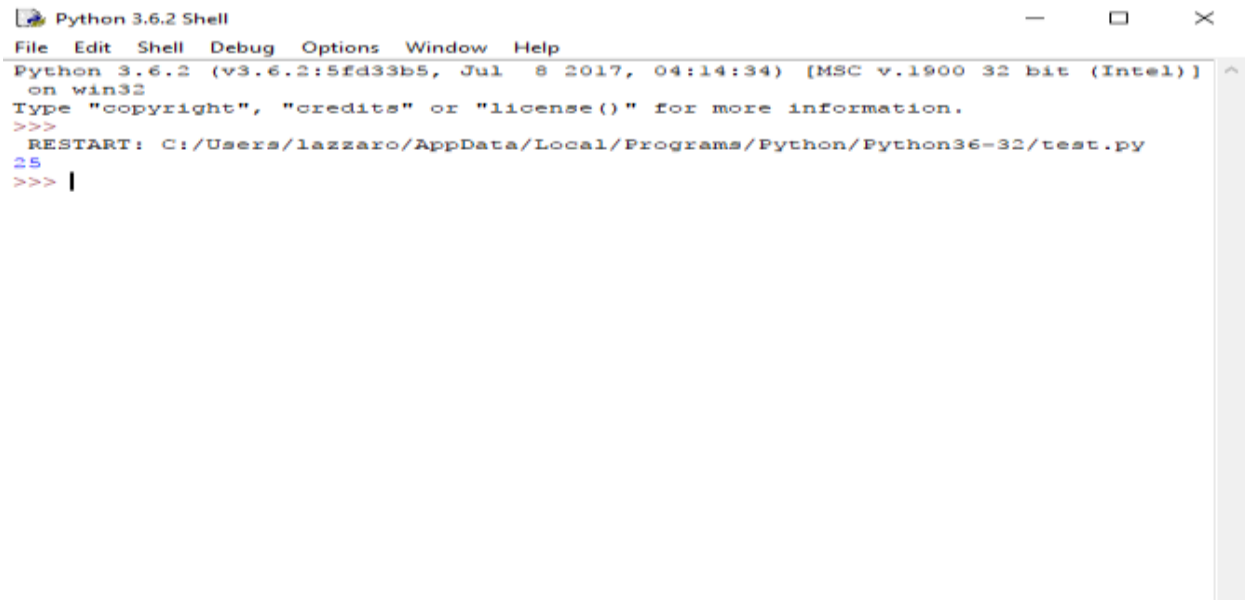


Click on file -> save as



- Run the code by clicking on Run in the Menu bar.
- Run -> Run Module

Result will be displayed on a new Python shell as:



```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:/Users/lazzaro/AppData/Local/Programs/Python/Python36-32/test.py
25
>>> |
```

2.2 Values and types

- A **value** is one of the basic things a program works with, like a letter or a number
- The values belong to different **types**:
 - 2 is an **integer**,
 - 42.0 is a **floating-point number**,
 - 'Hello, World!' is a **string**
- If you are not sure what type a value has, the interpreter can tell you:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

- In these results, the word “**class**” is used in the sense of a *category*;
- a **type** is a *category of values*.
 - integers belong to the type int,
 - strings belong to str
 - floating-point numbers belong to float.

- In python anything that is quoted in ' ' is considered as string
- '2' and '42.0'? They look like numbers, but they are in quotation marks like strings. Hence the type of '2' will be string.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
They're strings.
```

- We might be tempted to *use commas* between groups of *digits*, as in 1,000,000.
- This is *not a legal* integer in Python.

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

- Python interprets 1,000,000 as a *comma-separated sequence of integers*.
- This is the *example* we have seen of a *semantic error*:
- The code runs without producing an error message, but it doesn't do the "right" thing.

Integer

- Numbers are created by numeric literals.
- Numeric objects are immutable, which means when an object is created its value cannot be changed.
- Python has three distinct numeric types
 1. integers
 2. floating point numbers
 3. complex numbers.
- Integers represent negative and positive integers without fractional parts whereas floating point numbers represents negative and positive numbers with fractional parts.

```
>>> a=1452
>>> type(a)
<class 'int'>
>>> b=(-4587)
>>> type(b)
<class 'int'>
>>> c=0
>>> type(c)
<class 'int'>
>>> g=1.03
>>> type(g)
<class 'float'>
>>> h=-11.23
>>> type(g)
<class 'float'>
>>> i=.34
>>> type(i)
<class 'float'>
>>> j=2.12e-10
>>> type(j)
<class 'float'>
>>> k=5E220
>>> type(k)
<class 'float'>
>>>
```

Complex number

```
>>> x = complex(1,2)
>>> type(x)
<class 'complex'>
>>> print(x)
(1+2j)
>>> z = 1+2j
>>> type(z)
<class 'complex'>
>>> z = 1+2J
>>> type(z)
<class 'complex'>
>>> |
```

Boolean:

```
>>> x = True
>>> type(x)
<class 'bool'>
>>> y = False
>>> type(y)
<class 'bool'>
>>>
```

String

```
>>> str1 = "String" #Strings start and end with double quotes
>>> print(str1)
String
>>> str2 = 'String' #Strings start and end with single quotes
>>> print(str2)
String
>>> str3 = "String' #Strings start with double quote and end wit
h single quote
SyntaxError: EOL while scanning string literal
>>> str1 = 'String" #Strings start with single quote and end wit
h double quote
SyntaxError: EOL while scanning string literal
SyntaxError: EOL while scanning string literal
>>> str2 = "Day's" #Single quote within double quotes
>>> print(str2)
Day's
>>> str2 = 'Day"s' #Double quote within single quotes
>>> print(str2)
Day"s
>>> |
```

Special characters in strings

- The backslash (\) character is used to introduce a special character.

Escape sequence	Meaning
\n	Newline
\t	Horizontal Tab
\\	Backslash
\'	Single Quote
\"	Double Quote

```
The is a backslash (\) mark.
>>> print("This is tab \t key")
This is tab      key
>>> print("These are \'single quotes\'")
These are 'single quotes'
>>> print("These are \"double quotes\"")
These are "double quotes"
>>> print("This is a new line\nNew line")
This is a new line
New line
>>>
```

List

- A list is a container which holds comma-separated values (items or elements) between square brackets where Items or elements need not all have the same type.

```
>>> print(my_list1)
[5, 12, 13, 14]
>>> my_list2 = ['red', 'blue', 'black', 'white'] # the list contains all string values
>>> print(my_list2)
['red', 'blue', 'black', 'white']
>>> my_list3 = ['red', 12, 112.12] # the list contains a string, an integer and a float values
>>> print(my_list3)
['red', 12, 112.12]
>>> |
```

2.3 Variables

- One of the most powerful features of a programming language is the ability to manipulate **variables**.
- A variable is a *name* that *refers to a value*.

2.3.1 Assignment statements

- An **assignment statement** creates a new variable and gives it a value:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
```

- This example makes three assignments.
- The first assigns a string to a new variable named message;
- The second gives the integer 17 to n;
- The third assigns the (approximate) value of p to pi.

2.3.2 Variable names

- Programmers generally choose names for their variables that are *meaningful*
- Variable names can be as long as you like.
- They can contain *both letters and numbers*, but they can't begin with a number.
- It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.
- The *underscore* character, `_`, can appear in a name.
- It is often used in names with multiple words, such as `your_name` or `airspeed_of_unladen_swallow`.
- If you give a variable an *illegal name*, you get a *syntax error*:

```
>>> 76trombones = 'big parade'

SyntaxError: invalid syntax

>>> more@ = 1000000

SyntaxError: invalid syntax

>>> class = 'Advanced Theoretical Zymurgy'
```

SyntaxError: invalid syntax

- `76trombones` is illegal because it begins with a number.
- `more@` is illegal because it contains an illegal character, `@`.
- But what's wrong with `class`? -> It turns out that `class` is one of Python's **keywords**.
- The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names.

2.4 Expressions

- An **expression** is a combination of *values, variables, and operators*.
- A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 42
42
>>> n
```

```
17
>>> n + 25
42
```

- When you type an expression at the prompt, the interpreter **evaluates** it
- This means that it finds the value of the expression.
- In this example, `n` has the value 17 and `n + 25` has the value 42.

2.5 Statements

- A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

- The first line is an assignment statement that gives a value to `n`.
- The second line is a print statement that displays the value of `n`.
- When you type a statement, the interpreter **executes** it, which means that it does whatever the statement says.
- In general, statements don't have values.

2.6 Tuple assignment

- One of the most common operations done in computation is swapping.
- With conventional assignments, you have to use a temporary variable.
- For example, to swap `a` and `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

- This solution is burdensome;
- To avoid this usage of an additional variable called `temp` we can go for **tuple assignment** which is more elegant:

```
>>> a, b = b, a
```

- The left side is a tuple of variables;
- The right side is a tuple of expressions.
- Each value is assigned to its respective variable.
- All the expressions on the right side are evaluated before any of the assignments.
- The number of variables on the left and the number of values on the right have to be the Same

```
>>> a, b = 1, 2, 3
```

- This returns an **ValueError**: too many values to unpack.
- More generally, the right side can be any kind of sequence (string, list or tuple).
- For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

- The return value from `split` is a list with two elements;
- The first element is assigned to `uname`, the second to `domain`.

```
>>> print uname
monty
>>> print domain
python.org
```

2.7 Precedence of Operators

- When an expression contains more than one operator, the order of evaluation depends on the **order of operations**.
- For mathematical operators, Python follows mathematical convention.
- The acronym **PEMDAS** is a useful way to remember the rules:
 1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, `2 * 3 + 4`

(3-1) is 4, and $(1+1)**(5-2)$ is 8. You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even if it doesn't change the result.

2. **Exponentiation** has the next highest precedence, so $1 + 2**3$ ($1 + 2*2*2 = 1+8$) is 9, not 27, and $2 * 3**2$ ($2* 3*3$) is 18, not 36.
3. **Multiplication and Division** have higher precedence than **Addition and Subtraction**. So $2*3-1$ ($6-1$) is 5, not 4, and $6+4/2$ ($6+2$) is 8, not 5.
4. Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression $\text{degrees} / 2 * \text{pi}$, the division happens first and the result is multiplied by pi. To divide by 2π , you can use parentheses or write $\text{degrees} / 2 / \text{pi}$.

2.8 Comments

- As programs get bigger and more complicated, they get more difficult to read.
- Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
- For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing.
- These notes are called **comments**, and they start with the **#** symbol:

```
>>> # compute the percentage of the hour that has elapsed
>>> percentage = (minute * 100) / 60
```

- In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

```
>>> percentage = (minute * 100) / 60 # percentage of an hour
```

- Everything from the **#** to the end of the line is ignored
- It has no effect on the execution of the program.
- It is reasonable to assume that the reader can figure out what the code does.
- It is more useful to explain why this comment is redundant with the code and useless

```
>>> v = 5 # assign 5 to v
```

- This comment contains useful information that is not in the code:

```
>>> v = 5 # velocity in meters/second.
```

- Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a tradeoff.

2.9 Modules and Functions

2.9.1 Module

- Modules are used to *categorize code* in Python into smaller part.
- A module is simply a file, where *classes, functions and variables* are defined.
- Grouping *similar code into a single file* makes it easy to access.
- Python modules are the files which have similar code.

Advantage:

Python provides the following advantages for using module:

- 1) **Reusability:** Module can be used in some other python code. Hence it provides the facility of code reusability.
- 2) **Categorization:** Similar type of attributes can be placed in one module.

2.9.1.1 Importing a Module:

- There are different ways by which you we can import a module. These are as follows:

1) Using import statement:

- "import" statement can be used to import a module.

Syntax:

```
>>> import <file_name1, file_name2,...file_name(n)="">
>>> </file_name1,>
```

Have a look over an example:

```
>>> def add(a,b):  
>>> c=a+b  
>>> print c  
>>> return
```

- Save the file by the name ***addition.py***.
- To import this file "import" statement is used.

```
>>> import addition  
>>> addition.add(10,20)  
>>> addition.add(30,40)
```

- Create another python file in which you want to import the former python file.
- For that, import statement is used as given in the above example.
- The corresponding method can be used by ***file_name.method ()***. (Here, addition. add (), where addition is the python file and add () is the method defined in the file addition.py)

Output:

```
>>>  
30  
70  
>>>
```

- You can access any function which is inside a module by module name and function name separated by dot.
- It is also known as period.
- Whole notation is known as dot notation.

Example of importing multiple modules:

1) msg.py:

```
>>> def msg_method():
```

```
>>> print "Today the weather is rainy"
> return
```

2) display.py:

```
>>> def display_method():
>>> print "The weather is Sunny"
>>> return
```

3) multiimport.py:

```
>>> import msg,display
>>> msg.msg_method()
>>> display.display_method()
```

Output:

```
>>>
Today the weather is rainy
The weather is Sunny
>>>
```

2) Using from.. import statement:

- *from..import* statement is used to import *particular attribute* from a module.
- In case you do not want whole of the module to be imported then you can use from import statement.

Syntax:

```
from <module_name> import <attribute1,attribute2,attribute3,...attributen>
    </attribute1,attribute2,attribute3,...attributen></module_name>
```

Have a look over the example:

1) area.py

```
def circle(r):  
    print 3.14*r*r  
    return  
  
def square(l):  
    print l*l  
    return  
  
def rectangle(l,b):  
    print l*b  
    return  
  
def triangle(b,h):  
    print 0.5*b*h  
    return
```

2) area1.py

```
from area import square,rectangle  
square(10)  
rectangle(2,5)
```

Output:

```
>>>  
100  
10  
>>>
```

3) To import whole module:

- You can import *whole of the module* using "*from? import **"

Syntax:

```
from <module_name> import *  
</module_name>
```

- Using the above statement all the attributes defined in the module will be imported and hence you can access each attribute in that module.

1) area.py

```
def circle(r):  
print 3.14*r*r  
return  
  
def square(l):  
print l*l  
return  
  
def rectangle(l,b):  
print l*b  
return  
  
def triangle(b,h):  
print 0.5*b*h  
return
```

2) area1.py

```
from area import *  
square(10)  
rectangle(2,5)  
circle(5)  
triangle(10,20)
```

Output:

```
>>>  
100  
10  
78.5  
100.0  
>>>
```

2.9.1.2 Built-in Modules in Python:

- There are many built in modules in Python. Some of them are as follows:
math, random , threading , collections , os , mailbox , string , time , tkinter etc..
- Each module has a number of *built in functions* which can be used to perform various functions.

1) math:

- Using math module , you can use different built in *mathematical functions*.

Functions:

Function	Description
ceil(n)	Returns the next integer number of the given number
sqrt(n)	Returns the Square root of the given number.
exp(n)	Returns the natural logarithm e raised to the given number
floor(n)	Returns the previous integer number of the given number.
log(n,baseto)	Returns the natural logarithm of the number.
pow(baseto, exp)	Returns baseto raised to the exp power.
sin(n)	Returns sine of the given radian.
cos(n)	Returns cosine of the given radian.
tan(n)	Returns tangent of the given radian.

Useful Example of math module:

```
import math
a=4.6
print math.ceil(a)
print math.floor(a)
b=9
print math.sqrt(b)
print math.exp(3.0)
print math.log(2.0)
print math.pow(2.0,3.0)
```

```
print math.sin(0)
print math.cos(0)
print math.tan(45)
```

Output:

```
>>>
5.0
4.0
3.0
20.0855369232
0.69314718056
8.0
0.0
1.0
1.61977519054
>>>
```

Constants:

- The math module provides two constants for mathematical Operations:

Constants	Descriptions
Pi	Returns constant $\pi = 3.14159...$
ceil(n)	Returns constant $e = 2.71828...$

```
import math
print math.pi
print math.e
```

Output:

```
>>>
3.14159265359
2.71828182846
>>>
```

2)Random:

- The random module is used to generate the random numbers.

- It provides the following two built in functions:

Function	Description
random()	It returns a random number between 0.0 and 1.0 where 1.0 is exclusive.
randint(x,y)	It returns a random number between x and y where both the numbers are inclusive.

```
import random
```

```
print random.random()
```

```
print random.randint(2,8)
```

Output:

```
>>>
```

```
0.797473843839
```

```
7
```

```
>>>
```

2.9.1.3 Package

- A Package is simply a *collection of similar modules*, sub-packages etc..

Steps to create and import Package:

- 1) Create a directory,
- 2) Place different modules inside the directory.
- 3) Place corresponding codes in respective modules.
- 4) Create a file `__init__.py` which specifies attributes in each module.
- 5) Import the package and use the attributes using package

- For example create a directory say **Info**
- placing 3 modules `msg1.py`, `msg2.py` and `msg3.py` respectively
- Placing corresponding codes in respective modules as place `msg1()` in `msg1.py`, `msg2()` in `msg2.py` and `msg3()` in `msg3.py`.

1) Create the directory:

```
>>> import os
```

```
>>> os.mkdir("Info")
```

2) Place different modules in package: (Save different modules inside the Info package)

msg1.py

```
def msg1():  
    print "This is msg1"
```

msg2.py

```
def msg2():  
    print "This is msg2"
```

msg3.py

```
def msg3():  
    print "This is msg3"
```

3) Create __init__.py file:

```
from msg1 import msg1  
from msg2 import msg2  
from msg3 import msg3
```

4) Import package and use the attributes:

```
import Info  
Info.msg1()  
Info.msg2()  
Info.msg3()
```

Output:

```
>>>
This is msg1
This is msg2
This is msg3
>>>
```

- `__init__.py` is simply a file that is used to consider the directories on the disk as the package of the Python.
- It is basically used to *initialize* the python packages.

2.9.2 Functions

- A Function is a *self-block* of code.
- A Function can be called as a *section of a program* that is written once and can be executed whenever required in the program, thus making *code reusability*.
- A Function is a subprogram that works on data and produces some output.
- In the context of programming, a **function** is a named sequence of statements that performs a computation.
- When you define a function, you specify the name and the sequence of statements.
- Later, you can “call” the function by name.

```
>>> type(42)
<class 'int'>
```

- The name of the function is `type`.
- The expression in parentheses is called the **argument** of the function.
- The result, for this function, is the type of the argument.
- It is common to say that a function “takes” an argument and “returns” a result.
- The result is also called the **return value**.
- Python provides functions that *convert values from one type to another*.
- The `int` function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
```

```
ValueError: invalid literal for int(): Hello
```

- int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

- float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

- Finally, str converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Types of Functions:

- There are two types of Functions.
 - a) **Built-in Functions:** Functions that are predefined. We have many predefined functions in Python.
 - b) **User- Defined:** Functions that are created according to the requirements.

2.9.2.1 Return Statement:

- return[expression] is used to *send back the control to the caller* with the expression.
- In case no expression is given after return it will return None.
- In other words return statement is used to *exit the Function definition*.

Example:

```
def sum(a,b):
```

```

        "Adding the two values"
    print "Printing within Function"
print a+b
    return a+b
def msg():
    print "Hello"
    return

total=sum(10,20)
print ?Printing Outside: ?,total
msg()
print "Rest of code"

```

Output:

```

>>>
Printing within Function
30
Printing outside: 30
Hello
Rest of code
>>>

```

2.9.2.2 Scope of Variable:

- Scope of a variable can be determined by the part in which variable is defined.
- Each variable cannot be accessed in each part of a program.
- There are two types of variables based on Scope:

1) Local Variable.

2) Global Variable.

1) Local Variables:

- Variables declared *inside a function body* is known as Local Variable.

- These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Example:

```
def msg():  
    a=10  
    print "Value of a is",a  
    return  
  
msg()  
print a #it will show error since variable is local  
  
Output:  
  
>>>  
Value of a is 10  
  
Traceback (most recent call last):  
  File "C:/Python27/lam.py", line 7, in <module>  
    print a #it will show error since variable is local  
NameError: name 'a' is not defined  
>>>  
</module>
```

b) Global Variable:

- Variable defined *outside the function* is called Global Variable.
- Global variable is accessed all over program thus global variable have widest accessibility.

Example:

```
b=20  
def msg():  
    a=10
```

```
print "Value of a is",a
print "Value of b is",b
return
```

```
msg()
print b
```

Output:

```
>>>
Value of a is 10
Value of b is 20
20
>>>
```

2.10 Function definition and use

2.10.1 Defining a Function:

- A **function definition** specifies the name of a new function and the sequence of statements that run when the function is called.

The rules for function

- Rules for names are the same as for variable names: letters, numbers and underscore are legal
- The first character of function name can't be a number.
- You can't use a keyword as the name of a function
- Should avoid having a variable and a function with the same name.

Here is an example:

```
>>>def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

- *def* is a **keyword** that indicates that this is a function definition.
- The name of the function is `print_lyrics`.

- The empty parentheses after the name indicate that this function doesn't take any arguments.
- The first line of the function definition is called the **header**; the rest is called the **body**.
- The header has to end with a colon and the body has to be indented.
- By convention, indentation is always four spaces. The body can contain any number of statements
- The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing;
- Defining a function creates a **function object**, which has type function:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

- The syntax for calling the new function is the same as for built-in functions:

```
>>> print_lyrics()
```

Output

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

- Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write a function called `repeat_lyrics`:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

- And then call `repeat_lyrics`:

```
>>> repeat_lyrics()
```

Output

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

I'm a lumberjack, and I'm okay.

I sleep all night and I work all day.

But that's not really how the song goes.

➤ A Function defined in Python should follow the following format:

- 1) Keyword def is used to start the Function Definition. Def specifies the starting of Function block.
- 2) def is followed by function-name followed by parenthesis.
- 3) Parameters are passed inside the parenthesis. At the end a colon is marked.

Syntax:

```
def <function_name>([parameters]):  
</function_name>
```

Example:

```
def sum(a,b):
```

- 4) Before writing a code, an Indentation (space) is provided before every statement. It should be same for all statements inside the function.
- 5) The first statement of the function is optional. It is ?Documentation string? of function.
- 6) Following is the statement to be executed.

```
def keyword      Function name      Parameters  
def <function name>([parameters]):  
    "function docstring"  
    Statement1  
    Statement2  
    ...  
    ....  
Indentation
```

2.10.2 Invoking a Function:

- To execute a function it needs to be called. This is called *function calling*.
- Function Definition provides the information about function name, parameters and the definition what operation is to be performed.
- In order to execute the Function Definition it is to be called.
- Function call will be executed in the order in which it is called.

Syntax:

```
<function_name>(parameters)
</function_name>
```

Example: sum(a,b)

- Here sum is the function and a, b are the parameters passed to the Function Definition.

Example:

```
#Providing Function Definition
def sum(x,y):
    "Going to add x and y"
    s=x+y
    print "Sum of two numbers is"
    print s
#Calling the sum Function
sum(10,20)
sum(20,30)
```

Output:

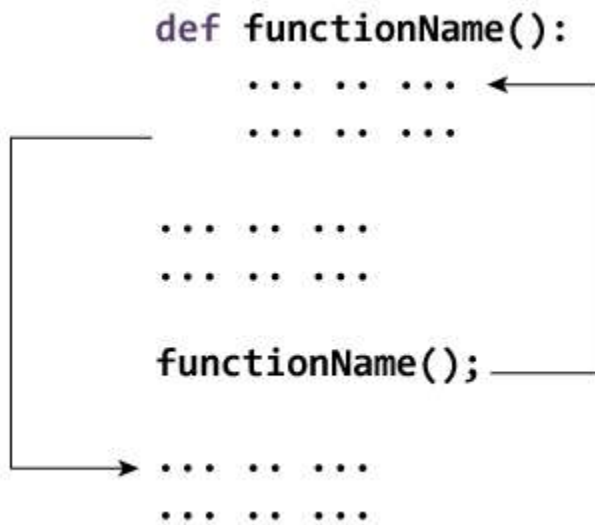
```
>>>
Sum of two numbers is
30
Sum of two numbers is
50
>>>
```

Advantages and need for functions

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

2.11 Flow of execution

- In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the **flow of execution**.
- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off.
- That sounds simple enough, until you remember that one function can call another.
- While in the middle of one function, the program might have to execute the statements in another function.
- But while executing that new function, the program might have to execute yet another function!
- Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it.
- When it gets to the end of the program, it terminates.
- When you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.



2.12 Parameters and Arguments

2.12.1 Argument and Parameter

➤ There can be two types of data passed in the function.

1) The First type of data is the data *passed in the function call*. This data is called arguments?

2) The second type of data is the data *received in the function definition*. This data is called parameters?.

- Arguments can be literals, variables and expressions.
- Parameters must be variable to hold incoming values.
- Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

Example:

```
def addition(x,y):  
    print x+y  
x=15  
addition(x ,10)
```

```
addition(x,x)
y=20
addition(x,y)
```

Output:

```
>>>
25
30
35
>>>
```

Passing Parameters

- Apart from matching the parameters, there are other ways of matching the parameters.
- Python supports following types of formal argument:

1) Positional argument (Required argument).

2) Default argument.

3) Keyword argument (Named argument)

Positional/Required Arguments:

- When the function call statement must match the number and order of arguments as defined in the function definition it is ***Positional Argument matching***.

Example

```
#Function definition of sum
def sum(a,b):
    "Function having two parameters"
    c=a+b
    print c

sum(10,20)
sum(20)
```

Output:

```
>>>
```

```
30
```

Traceback (most recent call last):

File "C:/Python27/su.py", line 8, in <module>

sum(20)

TypeError: sum() takes exactly 2 arguments (1 given)

```
>>>
```

```
</module>
```

Explanation:

1) In the first case, when sum() function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and printed.

2) In the second case, when sum() function is called passing a single value i.e., 20 , it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

Default Arguments

- Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.

Example

```
#Function Definition
```

```
def msg(Id,Name,Age=21):
```

```
    "Printing the passed value"
```

```
    print Id
```

```
    print Name
```

```
    print Age
```

```
    return
```

```
#Function call
```

```
msg(Id=100,Name='Ravi',Age=20)
```

```
msg(Id=101,Name='Ratan')
```

Output:

```
>>>
100
Ravi
20
101
Ratan
21
>>>
```

Explanation:

1) In first case, when msg() function is called passing three different values i.e., 100 , Ravi and 20, these values will be assigned to respective parameters and thus respective values will be printed.

2) In second case, when msg() function is called passing two values i.e., 101 and Ratan, these values will be assigned to Id and Name respectively. No value is assigned for third argument via function call and hence it will retain its default value i.e, 21.

Keyword Arguments:

- Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

Example

```
def msg(id,name):
    "Printing passed value"
    print id
    print name
    return
msg(id=100,name='Raj')
msg(name='Rahul',id=101)
```

Output:

```
>>>
```

```
100
Raj
101
Rahul
>>>
```

Explanation:

1) In the first case, when msg() function is called passing two values i.e., id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.

2) In second case, when msg() function is called passing two values i.e., name and id, although the position of two parameters is different it initialize the value of id in Function call to id in Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

2.12.2. Anonymous Function:

- Anonymous Functions are the functions that are *not bond to name*.
- Anonymous Functions are created by using a keyword "*lambda*".
- Lambda takes any number of arguments and returns an evaluated expression.
- Lambda is created *without using the def keyword*.

Syntax:

```
lambda arg1,args2,args3,?,argsn :expression
```

Example:

```
#Function Definiton
square=lambda x1: x1*x1

#Calling square as a function
print "Square of number is",square(10)
```

Output:


```
>>>
Square of number is 100
>>>
```

Difference between Normal Functions and Anonymous Function:

Example:

Normal function:

```
#Function Definiton
def square(x):
    return x*x

#Calling square function
print "Square of number is",square(10)
```

Anonymous function:

```
#Function Definiton
square=lambda x1: x1*x1

#Calling square as a function
print "Square of number is",square(10)
```

Explanation:

- Normal Function is created without using def keyword.
- lambda keyword is used to create anonymous function.
- It returns the evaluated expression.

2.13 Illustrative programs

2.13.1 Exchange the values of two variables

```

# Python program to swap two variables
# To take input from the user
# x = input('Enter value of x: ')
# y = input('Enter value of y: ')
x = 5
y = 10
# create a temporary variable and swap the values
temp = x
x = y
y = temp
print('The value of x after swapping: {}'.format(x))
print('The value of y after swapping: {}'.format(y))

```

2.13.2 Circulate the values of n variables

```

def shift_left(lst, n):
    """Shifts the lst over by n indices

    >>> lst = [1, 2, 3, 4, 5]
    >>> shift_left(lst, 2)
    >>> lst
    [3, 4, 5, 1, 2]
    """
    assert (n > 0), "n should be non-negative integer"
    def shift(ntimes):
        if ntimes == 0:
            return
        else:
            temp = lst[0]
            for index in range(len(lst) - 1):
                lst[index] = lst[index + 1]
            lst[index + 1] = temp
            return shift(ntimes-1)
    return shift(n)

```

2.13.3 Distance between two points

```

import math

p1 = [4, 0]
p2 = [6, 6]
distance = math.sqrt( ((p1[0]-p2[0])**2)+((p1[1]-p2[1])**2) )
print(distance)

```

Part A

1. Define Python interpreter?

The Python *interpreter* is a program that reads and *executes Python code*. Depending on your environment, you might start the interpreter by clicking on an icon, or by typing python on a command line.

2. What are the Basic modes of python?

Python has two basic modes: normal and interactive. The normal mode is the mode where the scripted and finished .py files are run in the Python interpreter. Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory. As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole. Interactive mode is a good way to play around and try variations on syntax.

3. Define values?

A **value** is one of the basic things a program works with, like a letter or a number. The values belong to different **types**:

2 is an **integer**,
42.0 is a **floating-point number**,
'Hello, World!' is a **string**

4. Define strings?

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

5. What are the special characters used in strings?

Escape sequence	Meaning
\n	Newline
\t	Horizontal Tab
\\	Backslash
\'	Single Quote
\"	Double Quote

6. Define list.

A list is a container which holds comma-separated values (items or elements) between square brackets where Items or elements need not all have the same type.

7. Define Tuples?

A tuple is another sequence data type that is similar to the list. A tuple is a sequence of immutable Python objects. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

8. Difference between lists and tuples?

The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated. Tuples can be thought of as **read-only** lists.

9. Define variables.

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a *name* that *refers to a value*.

Example: a=5

10. Define Expressions?

An **expression** is a combination of *values, variables, and operators*.

11. What are the types of operators used in python?

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

12. Mention the precedence of the operators from the highest to the lowest?

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division

+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

13. Define module.

Modules are used to *categorize code* in Python into smaller part. A module is simply a file, where *classes, functions and variables* are defined. Grouping *similar code into a single file* makes it easy to access. Python modules are the files which have similar code.

14. What are the advantages of module?

Python provides the following advantages for using module:

- 1) **Reusability:** Module can be used in some other python code. Hence it provides the facility of code reusability.
- 2) **Categorization:** Similar type of attributes can be placed in one module.

15. What are the built-in modules in python?

There are many built in modules in Python. Some of them are as follows: **math, random , threading , collections , os , mailbox , string , time , tkinter etc..**

16. Define package.

A Package is simply a *collection of similar modules*, sub-packages etc..

17. What are the steps to create and import package?

- 1) Create a directory,
- 2) Place different modules inside the directory.
- 3) Place corresponding codes in respective modules.
- 4) Create a file `__init__.py` which specifies attributes in each module.
- 5) Import the package and use the attributes using package

18.What is a method

Methods are simply another kind of function that resides in classes. You create and work with methods in Python in precisely the same way that you do functions, except that methods are always associated with a class.

19 what are the kinds of methods:

- 1.those associated with the class itself a
2. those associated with an instance of a class.

20. what is a class method?

A *class method* is one that you execute directly from the class without creating an instance of the class. Sometimes you need to create methods that execute from the class.

21.What is a instance method.

An *instance method* is one that is part of the individual instances. You use instance methods to manipulate the data that the class manages. As a consequence, you can't use instance methods until you instantiate an object from the class. All instance methods accept a single argument as a minimum, self. The self argument points at the particular instance that the application is using to manipulate data. Without the self argument, the method wouldn't know which instance data to use. However, self isn't considered an accessible argument — the value for self is supplied by Python, and you can't change it as part of calling the method.

Part B

1. What is meant by Python interpreter? Explain the various modes in detail.
2. What are the values and types in python. Discuss in detail.
3. Explain variables and its types.
4. Discuss about the expressions, statements, tuple assignment and precedence of operators
5. Define the term comments. Discuss briefly.
6. What are the modules and functions. Explain in detail about function definition and use,
7. Discuss the flow of execution
8. Explain the parameters and arguments briefly.
9. Illustrative programs: exchange the values of two variables
10. Illustrative programs: circulate the values of n variables,
11. Illustrative programs: distance between two points.

Exercises (Case study)

12. We've seen that $n = 42$ is legal. What about $42 = n$?
13. How about $x = y = 1$?
14. In some languages every statement ends with a semi-colon, ;. What happens if you put a semi-colon at the end of a Python statement?
15. What if you put a period at the end of a statement?
16. In math notation you can multiply x and y like this: xy . What happens if you try that in Python?
17. The volume of a sphere with radius r is $\frac{4}{3}\pi r^3$. What is the volume of a sphere with radius 5?
18. Suppose the cover price of a book is \$24.95, but bookstores get a 40% discount. Shipping costs \$3 for the first copy and 75 cents for each additional copy. What is the total wholesale cost for 60 copies?
19. If I leave my house at 6:52 am and run 1 mile at an easy pace (8:15 per mile), then 3 miles at tempo (7:12 per mile) and 1 mile at easy pace again, what time do I get home for breakfast?
20. Write a function named `right_justify` that takes a string named `s` as a parameter and prints the string with enough leading spaces so that the last letter of the string is in column 70 of the display.

```
>>> right_justify('monty')
monty
```

Hint: Use string concatenation and repetition. Also, Python provides a built-in function called `len` that returns the length of a string, so the value of `len('monty')` is 5.

21. A function object is a value you can assign to a variable or pass as an argument. For example, `do_twice` is a function that takes a function object as an argument and calls it twice:

```
def do_twice(f):
    f()
    f()
```

Here's an example that uses `do_twice` to call a function named `print_spam` twice.

```
def print_spam():
    print('spam')
do_twice(print_spam)
```

1. Type this example into a script and test it.
2. Modify `do_twice` so that it takes two arguments, a function object and a value, and calls the function twice, passing the value as an argument.
3. Copy the definition of `print_twice` from earlier in this chapter to your script.
4. Use the modified version of `do_twice` to call `print_twice` twice, passing 'spam' as an argument.
5. Define a new function called `do_four` that takes a function object and a value and calls the function four times, passing the value as a parameter. There should be only two statements in the body of this function, not four.

Solution: http://thinkpython2.com/code/do_four.py.

22. Note: This exercise should be done using only the statements and other features we have learned so far.

1. Write a function that draws a grid like the following:

```

+ - - - + - - - +
| | |
| | |
| | |
| | |
+ - - - + - - - +
| | |
| | |
| | |
| | |
+ - - - + - - - +

```

Hint: to print more than one value on a line, you can print a comma-separated sequence of values:

```
print('+', '-')
```

By default, print advances to the next line, but you can override that behavior and put a space at the end, like this:

```
print('+', end=' ')
```

```
print('-')
```

The output of these statements is '+ -' on the same line. The output from the next print statement would begin on the next line.

23. Write a function that draws a similar grid with four rows and four columns.

Solution: <http://thinkpython2.com/code/grid.py>. Credit: This exercise is based on an exercise in Oualline, Practical C Programming, Third Edition, O'Reilly Media, 1997.

24. In a print statement, what happens if you leave out one of the parentheses, or both?

25. If you are trying to print a string, what happens if you leave out one of the quotation marks, or both?

26. You can use a minus sign to make a negative number like -2. What happens if you put a plus sign before a number? What about 2++2?

27. In math notation, leading zeros are ok, as in 02. What happens if you try this in Python?

28. What happens if you have two values with no operator between them?

29. How many seconds are there in 42 minutes 42 seconds?

30. How many miles are there in 10 kilometers? Hint: there are 1.61 kilometers in a mile.

31. If you run a 10 kilometer race in 42 minutes 42 seconds, what is your average pace (time per mile in minutes and seconds)? What is your average speed in miles per hour?

Glossary

variable: A name that refers to a value.

assignment: A statement that assigns a value to a variable.

state diagram: A graphical representation of a set of variables and the values they refer to.

keyword: A reserved word that is used to parse a program; you cannot use keywords like

if, def, and while as variable names.

operand: One of the values on which an operator operates.

expression: A combination of variables, operators, and values that represents a single result.

evaluate: To simplify an expression by performing the operations in order to yield a single value.

statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.

execute: To run a statement and do what it says.

interactive mode: A way of using the Python interpreter by typing code at the prompt.

script mode: A way of using the Python interpreter to read code from a script and run it.

script: A program stored in a file.

order of operations: Rules governing the order in which expressions involving multiple operators and operands are evaluated.

concatenate: To join two operands end-to-end.

comment: Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).

exception: An error that is detected while the program is running.

semantics: The meaning of a program.

semantic error: An error in a program that makes it do something other than what the programmer intended.

Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.

Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so $(1 + 2)$ is legal, but $8)$ is a **syntax error**.

If there is a syntax error anywhere in your program, Python displays an error message

and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic error: The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function definition: A statement that creates a new function, specifying its name, parameters, and the statements it contains.

function object: A value created by a function definition. The name of the function is a variable that refers to a function object.

header: The first line of a function definition.

body: The sequence of statements inside a function definition.

parameter: A name used inside a function to refer to the value passed as an argument.

function call: A statement that runs a function. It consists of the function name followed by an argument list in parentheses.

argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

local variable: A variable defined inside a function. A local variable can only be used inside its function.

return value: The result of a function. If a function call is used as an expression, the return value is the value of the expression.

fruitful function: A function that returns a value.

void function: A function that always returns None.

None: A special value returned by void functions.

module: A file that contains a collection of related functions and other definitions.

import statement: A statement that reads a module file and creates a module object.

module object: A value created by an import statement that provides access to the values defined in a module.

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

composition: Using an expression as part of a larger expression, or a statement as part of a larger statement.

flow of execution: The order statements run in.

stack diagram: A graphical representation of a stack of functions, their variables, and the values they refer to.

frame: A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.

traceback: A list of the functions that are executing, printed when an exception occurs.

problem solving: The process of formulating a problem, finding a solution, and expressing it.

high-level language: A programming language like Python that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to run; also called “machine language” or “assembly language”.

portability: A property of a program that can run on more than one kind of computer.

interpreter: A program that reads another program and executes it

prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

program: A set of instructions that specifies a computation.

print statement: An instruction that causes the Python interpreter to display a value on the screen.

operator: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

value: One of the basic units of data, like a number or string, that a program manipulates.

type: A category of values. The types we have seen so far are integers (type int), floatingpoint numbers (type float), and strings (type str).

integer: A type that represents whole numbers.

floating-point: A type that represents numbers with fractional parts.

string: A type that represents sequences of characters.

natural language: Any one of the languages that people speak that evolved naturally.

formal language: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

token: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

syntax: The rules that govern the structure of a program.

parse: To examine a program and analyze the syntactic structure.

bug: An error in a program.

debugging: The process of finding and correcting bugs.