**1.About Python**

**1.1 Python Overview**

* Open source
* General purpose
* Easy to interface with C/objC/Java/Fortran/C++
* Great interactive environment

Website and downloads: [www.python.org](http://www.python.org)

Documentation: [www.python.org](http://www.python.org)/doc/

**1.2 History of Python**

Innovative languages are mostly a product of either, a large well-funded research project or,frustration out of the lack of tools that were needed at the time to accomplish mundane or time taking tasks, where most of them could be automated could be automated.

* Conceived in the late 1980s
* Work on python began in 1989 by Guido van Rossum, at CWI in the Netherlands
* Made as a successor to ABC capable of exception handling and interfacing with the Amoeba operating system
* Released for public distribution in February 1991 labeled version 0.9.0,
* Already present at python version 0.9.0 in development were core features classes with inheritance, exception handling, functions, and the core data types of “list”, “dict” and “str” and so on.

**1.3 Features of Python**

* High level
* Object oriented
* Scalable
* Extensible
* Portable
* Easy to learn
* Easy to read
* Easy to maintain
* Robust
* Effective as a rapid prototyping tool
* A memory manager
* Interpreted and byte compiled

**1.4 Python Environment**

Before we start writing our programs in python , it’s important to know how to set up a Python environment. Python is available on a wide Number of platforms. Open a terminal window and type "python" to check if its already installed and which version you have if it is already installed.

* Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, et al.)
* Win 9x/NT/2000 (Windows 32-bit systems)
* Macintosh (PPC, 68K)
* OS/2
* DOS (multiple versions)
* Windows 3.x
* PalmOS
* Windows CE
* Acorn/RISC OS
* BeOS
* Amiga
* VMS/OpenVMS
* QNX
* VxWorks
* Psion
* Python is also ported to Java and .Net VM’s

**1.5 Getting Python**

For the most up-to-date and current source code, binaries, documentation, news, etc., check either the main Python language site or the PythonLabs Web site:

http://www.python.org (community home page)

http://www.pythonlabs.com (commercial home page)

If you do not have access to the Internet readily available, all three versions (source code and binaries) are available on the CD-ROM in the back of the book. The CD-ROM also features the complete online documentation sets viewable via offline browsing or as archive files which can be installed on hard disk. All of the code samples in the book are there as well as the Online Resources appendix section (featured as the Python "hotlist").

**1.6 Case studies**

**Exercise**

1. **Context of Software Developments**

A computer program, from one perspective, is a sequence of instructions that dictate the flow of electrical impulses within a computer system. These impulses affect the computer’s memory and interact with the display screen, keyboard, and mouse in such a way as to produce the “magic” that permits humans to perform useful tasks, solve high-level problems, and play games. One program allows a computer to assume the role of a financial calculator, while another transforms the machine into a worthy chess opponent. Note the two extremes here:

• at the lower, more concrete level electrical impulses alter the internal state of the computer, while

• at the higher, more abstract level computer users accomplish real-world work or derive actual pleasure.

So well is the higher-level illusion achieved that most computer users are oblivious to the lower-level activity (the machinery under the hood, so to speak). Surprisingly, perhaps, most programmers today write software at this higher, more abstract level also. An accomplished computer programmer can develop sophisticated software with little or no interest or knowledge of the actual computer system upon which it runs. Powerful software construction tools hide the lower-level details from programmers, allowing them to solve problems in higher-level terms.

The concepts of computer programming are logical and mathematical in nature. In theory, computer programs can be developed without the use of a computer. Programmers can discuss the viability of a program and reason about its correctness and efficiency by examining abstract symbols that correspond to the features of real-world programming languages but appear in no real-world programming language.While such exercises can be very valuable, in practice computer programmers are not isolated from their machines. Software is written to be used on real computer systems. Computing professionals known as software engineers develop software to drive particular systems. These systems are defined by their underlying hardware and operating system. Developers use concrete tools like compilers, debuggers, and profilers.

* 1. **Software**

A computer program is an example of computer software. One can refer to a program as a piece of software as if it were a tangible object, but software is actually quite intangible. It is stored on a medium. A hard drive, a CD, a DVD, and a USB pen drive are all examples of media upon which software can reside. The CD is not the software; the software is a pattern on the CD. In order to be used, software must be stored in the computer’s memory. Typically computer programs are loaded into memory from a medium like the computer’s hard disk. An electromagnetic pattern representing the program is stored on the computer’s hard drive. This pattern of electronic symbols must be transferred to the computer’s memory before the program can be executed. The program may have been installed on the hard disk from a CD or from the Internet. In any case, the essence that was transferred from medium to medium was a pattern of electronic symbols that direct the work of the computer system.

These patterns of electronic symbols are best represented as a sequence of zeroes and ones, digits from the binary (base 2) number system. An example of a binary program sequence is

10001011011000010001000001001110

To the underlying computer hardware, specifically the processor, a zero here and three ones there might mean that certain electrical signals should be sent to the graphics device so that it makes a certain part of the display screen red. Unfortunately, only a minuscule number of people in the world would be able to produce, by hand, the complete sequence of zeroes and ones that represent the program Microsoft Word

for an Intel-based computer running the Windows 7 operating system. Further, almost none of those who could produce the binary sequence would claim to enjoy the task.

The Word program for older Mac OS X computers using a PowerPC processor works similarly to the Windows version and indeed is produced by the same company, but the program is expressed in a completely different sequence of zeroes and ones! The Intel Core 2 Duo processor in the Windows machine accepts a completely different binary language than the PowerPC processor in the Mac. We say the processors have their own machine language.

* 1. **Development Tools**

If very few humans can (or want) to speak the machine language of the computers’ processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like Python allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, C++, Java, and C#. Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language.

One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called compilers that translate one computer language into another have been around for 60 years, but natural language processing is still an active area of artificial intelligence research. Natural languages, as they are used

by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today’s software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any program that can be solved by a computer.

Consider the following program fragment written in the Python programming language:

**subtotal = 25**

**tax = 3**

**total = subtotal + tax**

These three lines do not make up a complete Python program; they are merely a piece of a program. The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, subtotal, tax, and total, called variables, are used to hold information. Mathematicians have used variables for hundreds of years before the first digital computer was built. In programming, a variable represents a value stored in the computer’s memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the Python language, not machine language, it cannot be executed directly on any processor. A program called an interpreter translates the Python code into machine code when a user runs the program.

The higher-level language code is called source code. The interpreted machine language code is called the target code. The interpreter translates the source code into the target machine language.

The beauty of higher-level languages is this: the same Python source code can execute on different target platforms. The target platform must have a Python interpreter available, but multiple Python interpreters are available for all the major computing platforms. The human programmer therefore is free to think about writing the solution to the problem in Python, not in a specific machine language.

Programmers have a variety of tools available to enhance the software development process. Some common tools include:

**Editors**. An editor allows the programmer to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colors to highlight language features. The syntax of a language refers to the way pieces of the language are arranged to make well-formed sentences. To illustrate, the sentence

The tall boy runs quickly to the door.

uses proper English syntax. By comparison, the sentence

Boy the tall runs door to quickly the.

is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English.

Similarly, programming languages have strict syntax rules that must be followed to create well formed programs. Only well-formed programs are acceptable and can be compiled and executed. Some syntax-aware editors can use colors or other special annotations to alert programmers of syntax errors before the program is compiled.

**Compilers**. A compiler translates the source code to target code. The target code may be the machine language for a particular platform or embedded device. The target code could be another source language; for example, the earliest C++ compiler translated C++ into C, another higher-level language. The resulting C code was then processed by a C compiler to produce an executable program. (C++ compilers today translate C++ directly into machine language.)

Interpreters. An interpreter is like a compiler, in that it translates higher-level source code into machine language. It works differently, however. While a compiler produces an executable program that may run many times with no additional translation needed, an interpreter translates source code statements into machine language as the program runs. A compiled program does not need to be recompiled to run, but an interpreted program must be interpreted each time it is executed. In general, compiled programs execute more quickly than interpreted programs because the translation activity occurs only once. Interpreted programs, on the other hand, can run as is on any platform with an appropriate interpreter; they do not need to be recompiled to run on a different platform. Python, for example, is used mainly as an interpreted language, but compilers for it are available. Interpreted languages are better suited for dynamic, explorative development which many people feel is ideal for beginning programmers.

Debuggers. A debugger allows programmers to simultaneously run a program and see which source code line is currently being executed. The values of variables and other program elements can be watched to see if their values change as expected. Debuggers are valuable for locating errors (also called bugs) and repairing programs that contain errors. (See Section 3.4 for more information about programming errors.)

Profilers. A profiler is used to evaluate a program’s performance. It indicates how many times a portion of a program is executed during a particular run, and how long that portion takes to execute. Profilers also can be used for testing purposes to ensure all the code in a program is actually being used somewhere during testing. This is known as coverage. It is common for software to fail after its release because users exercise some part of the program that was not executed anytime during testing. The main purpose of profiling is to find the parts of a program that can be improved to make the program run faster.

Many developers use integrated development environments (IDEs). An IDE includes editors, debuggers, and other programming aids in one comprehensive program. Examples of commercial IDEs include Microsoft’s Visual Studio 2010, the Eclipse Foundation’s Eclipse IDE, and Apple’s XCode. IDLE is a very

simple IDE for Python.

1. **Installing Python**

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python. If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

* 1. **Unix & Linux Installation**

Here are the simple steps to install Python on Unix/Linux machine.

• Open a Web browser and go to www.python.org/download/

• Follow the link to download zipped source code available for Unix/Linux.

• Download and extract files.

• Editing the *Modules/Setup* file if you want to customize some options.

• **run** ./configure script

• make

• make install

This will install python in a standard location */usr/local/bin* and its libraries are installed in*/usr/local/lib/pythonXX* where XX is the version of Python that you are using.

* 1. **Windows Installation**

Here are the steps to install Python on Windows machine.

• Open a Web browser and go to www.python.org/download/

• Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you are going toinstall.

• To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Just save the installer file to your local machine and then run it to find out if your machine supports MSI.

• Run the downloaded file by double-clicking it in Windows Explorer. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you're ready to roll!

* 1. **Macintosh Installation**

Recent Macs come with Python installed, but it may be several years out of date. See

www.python.org/download/mac/ for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available."

Jack Jansen maintains it and you can have full access to the entire documentation at his Web site - **Jack Jansen**

**Website :** http://www.cwi.nl/~jack/macpython.html

.

* 1. **Setting up a path**

Programs and other executable files can live in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.The path is stored in an environment variable, which is a named string maintained by the operating system. These variables contain information available to the command shell and other programs. The **path** variable is named PATH in Unix or Path in Windows (Unix is case-sensitive; Windows is not). In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

* + 1. **Setting up a path at Unix/Linux**

To add the Python directory to the path for a particular session in Unix:

• **In the csh shell:** type

setenv PATH "$PATH:/usr/local/bin/python" and press Enter.

• **In the bash shell (Linux):** type

export PATH="$PATH:/usr/local/bin/python" and press Enter.

• **In the sh or ksh shell:** type

PATH="$PATH:/usr/local/bin/python" and press Enter.

**Note:** /usr/local/bin/python is the path of the Python directory

* + 1. **Setting up a path at Windows**

To add the Python directory to the path for a particular session in Windows:

• **At the command prompt :** type

path %path%;C:\Python and press Enter.

**Note:** C:\Python is the path of the Python directory

* 1. **Python Environment Variables**

Here are important environment variables, which can be recognized by Python:

|  |  |
| --- | --- |
| **Variable** | **Description** |
| PYTHONPATH | Has a role similar to PATH. This variable tells the Python interpreter where to locate the module files you import into a program. PYTHONPATH should include the Python source library directory and the directories containing your Python source code. PYTHONPATH is  sometimes preset by the Python installer. |
| PYTHONSTARTUP | Contains the path of an initialization file containing Python source code that is executed every time you start the interpreter (similar to the Unix .profile or .login file). This file, often named .pythonrc.py in Unix, usually contains commands that load utilities or modify PYTHONPATH. |
| PYTHONCASEOK | Used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it. |
| PYTHONHOME | An alternative module search path. It's usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy. |

1. **Running Python Programs**

There are three different ways to start Python:

* 1. **Interactive Interpreter**

You can enter **python** and start coding right away in the interactive interpreter by starting it from the command line. You can do this from Unix, DOS or any other system, which provides you a command-line interpreter or shell window.

|  |
| --- |
| $python # Unix/Linux  or  python% # Unix/Linux  or  C:>python # Windows/DOS |

Here is the list of all the available command line options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| -d | provide debug output |
| -O | generate optimized bytecode (resulting in .pyo files) |
| -S | do not run import site to look for Python paths on startup |
| -v | verbose output (detailed trace on import statements) |
| -X | disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6 |
| -c cmd | run Python script sent in as cmd string |
| file | run Python script from given file |

* 1. **Script from the command line**

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

|  |
| --- |
| $python script.py # Unix/Linux  or  python% script.py# Unix/Linux  or  C:>python script.py# Windows/DOS |

**Note:** Be sure the file permission mode allows execution.

* 1. **Integrated development Environment(IDE)**

You can run Python from a graphical user interface (GUI) environment as well. All you need is a GUI application on your system that supports Python.

• **Unix:** IDLE is the very first Unix IDE for Python.

• **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.

• **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

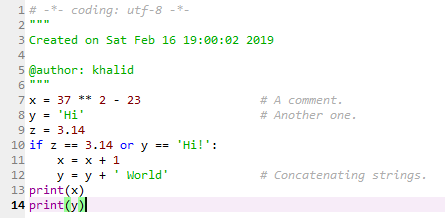
Before proceeding to next chapter, make sure your environment is properly set up and working perfectly fine. If you are not able to set up the environment properly, then you can take help from your system admin.

All the examples given in subsequent chapters have been executed with Python 2.4.3 version available on CentOS flavor of Linux.

1. **Python Basic Syntax**

Python is an interpreted language which means it is executed sequentially from the first line to the last line, and of course control flow can be changed by control flow statements.

**Sample python code**



* 1. **First Python Program**

There are two ways you can take to run your python code.

1. Interactive mode programming: Using the python shell to run single a statement at a time
2. Script mode programming: Running your python script as a whole.

**Interactive Mode Programming: (Using the Python shell )**

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:



Type the following text to the right of the Python prompt and press the Enter key:

|  |
| --- |
| >>> print("Hi World!") |

this will produce following result:

|  |
| --- |
| Hi World! |

**Script Mode Programming:**

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. All python files will have extension **.py**. So put the following source code in a test.py file.

|  |
| --- |
| print "Hello, Python!"; |

Here, I assumed that you have Python interpreter set in PATH variable. Now, try to run this program as follows:

|  |
| --- |
| $ python test.py |

This will produce the following result:

|  |
| --- |
| Hello, Python! |

Let's try another way to execute a Python script. Below is the modified test.py file:

|  |
| --- |
| #!/usr/bin/python  print "Hello, Python!"; |

Here, I assumed that you have Python interpreter available in /usr/bin directory. Now, try to run this program as

follows:

|  |
| --- |
| $ chmod +x test.py # This is to make file executable  $./test.py |

This will produce the following result:

|  |
| --- |
| Hello, Python! |

* 1. **Python Identifiers**

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_)followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $ and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are following identifier naming convention for Python:

• Class names start with an uppercase letter and all other identifiers with a lowercase letter.

• Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be

private.

• Starting an identifier with two leading underscores indicates a strongly private identifier.

• If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

* 1. **Reserved words**

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

|  |  |  |
| --- | --- | --- |
| And | Exec | Not |
| Assert | Finally | Or |
| Break | For | Pass |
| Class | From | Pass |
| Continue | Global | Raise |
| Def | If | Return |
| Del | Import | Try |
| Elif | In | While |
| Else | Is | With |
| Except | Lambda | Yield |

* 1. **Lines and Indentation**

One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

|  |
| --- |
| if True:  print "True"  else:  print "False" |

However, the second block in this example will generate an error:

|  |
| --- |
| if True:  print "Answer"  print "True"  else:  print "Answer"  print "False" |

Thus, in Python all the continous lines indented with similar number of spaces would form a block. Following is the example having various statement blocks:

**Note:** Don't try to understand logic or different functions used. Just make sure you understood various blocks even if they are without braces.

|  |
| --- |
| #!/usr/bin/python  import sys  try:  # open file stream  file = open(file\_name, "w")  except IOError:  print "There was an error writing to", file\_name  sys.exit()  print "Enter '", file\_finish,  print "' When finished"  while file\_text != file\_finish:  file\_text = raw\_input("Enter text: ")  if file\_text == file\_finish:  # close the file  file.close  break  file.write(file\_text)  file.write("\n")  file.close()  file\_name = raw\_input("Enter filename: ")  if len(file\_name) == 0:  print "Next time please enter something"  sys.exit()  try:  file = open(file\_name, "r")  except IOError:  print "There was an error reading file"  sys.exit()  file\_text = file.read()  file.close()  print file\_text |

* 1. **Multi-Line Statements**

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

|  |
| --- |
| total = item\_one + \  item\_two + \  item\_three |

Statements contained within the [], {} or () brackets do not need to use the line continuation character. For example:

|  |
| --- |
| days = ['Monday', 'Tuesday', 'Wednesday',  'Thursday', 'Friday'] |

* 1. **Quotation in python**

Python accepts single ('), double (") and triple (''' or """') quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

|  |
| --- |
| word = 'word'  sentence = "This is a sentence."  paragraph = """This is a paragraph. It is  made up of multiple lines and sentences.""" |

* 1. **Comments in python**

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

|  |
| --- |
| #!/usr/bin/python  # First comment  print "Hello, Python!"; # second comment |

This will produce the following result:

|  |
| --- |
| Hello, Python! |

A comment may be on the same line after a statement or expression:

|  |
| --- |
| name = "Madisetti" # This is again comment |

You can comment multiple lines as follows:

|  |
| --- |
| # This is a comment.  # This is a comment, too.  # This is a comment, too.  # I said that already. |

* 1. **Using blank Lines**

A line containing only white-space, possibly with a comment, is known as a blank line and Python totally ignores it. In an interactive interpreter session, you must enter an empty physical line to terminate a multi-line statement.

* 1. **Waiting for the users**

The following line of the program displays the prompt, Press the enter key to exit and waits for the user to press the Enter key:

|  |
| --- |
| #!/usr/bin/python  raw\_input("\n\nPress the enter key to exit.") |

Here, "\n\n" are being used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

* 1. **Multiple statements on a single line**

The semicolon ( ; ) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

|  |
| --- |
| import sys; x = 'foo'; sys.stdout.write(x + '\n') |

* 1. **Multiple statement groups as suites**

A group of individual statements, which make a single code block are called **suites** in Python. Compound or complex statements, such as if, while, def, and class, are those

which require a header line and a suite.

Header lines begin the statement (with the keyword) and terminate with a colon ( : ) and are followed by one or more lines, which make up the suite. For example:

|  |
| --- |
| if expression :  suite  elif expression :  suite  else :  suite |

* 1. **Command line arguments**

You may have seen, for instance, that many programs can be run so that they provide you with some basic information about how they should be run, Python lets you do this with -h:

|  |
| --- |
| $ python -h  usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...  Options and arguments (and corresponding environment variables):  -c cmd : program passed in as string (terminates option list)  -d : debug output from parser (also PYTHONDEBUG=x)  -E : ignore environment variables (such as PYTHONPATH)  -h : print this help message and exit  [ etc. ] |

You can also program your script in such a way that it should accept various options.

**5.12.1 Accessing command line arguments**

Python provides a **getopt** module that helps you parse command-line options and arguments.

|  |
| --- |
| $ python test.py arg1 arg2 arg3 |

The Python **sys** module provides access to any command-line arguments via the **sys.argv**. This serves two purpose:

* sys.argv is the list of command-line arguments.
* len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program ie. script name.

**Example:**

Consider a script myprog.py

|  |
| --- |
| #!/usr/bin/python  import sys  print('Number of arguments ', len(sys.argv), 'arguments.')  print('Argument List ', str(sys.argv)) |

When the above script is run as follows

|  |
| --- |
| $ python myprog.py arg1 arg2 arg3 |

Output you would get is

|  |
| --- |
| Number of arguments 4 arguments.  Argument List ['myprog.py', 'arg1', 'arg2', 'arg3'] |

Note that the script’s name is the first argument, and its also considered when counting the number of arguments

**5.12.2 Parsing command line arguments**

Python provided a **getopt** module that helps you parse command-line options and arguments. This module provides two functions and an exception to enable command-line argument parsing. This tutorial would discuss about one method and one exception, which are sufficient for your programming requirements.

**getopt.getopt( ) method:**

This method parses command-line options and parameter list. Following is simple syntax for this method:

|  |
| --- |
| getopt.getopt(args, options[, long\_options]) |

Here is the detail of the parameters:

* **args**: This is the argument list to be parsed.
* **options**: This is the string of option letters that the script wants to recognize, with options that require an argument should be followed by a colon ( : ).
* **long\_options**: This is optional parameter and if specified, must be a list of strings with the names of the long options, which should be supported. Long options, which require an argument should be followed by an equal sign ('='). To accept only long options, options should be an empty string.

This method returns value consisting of two elements: the first is a list of **(option, value)** pairs. The second is the list of program arguments left after the option list was stripped.

Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options

(e.g., '-x') or two hyphens for long options (e.g., '--long-option').

**Exception getopt.GetoptError :**

This is exception gets raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none.

The argument to the exception is a string indicating the cause of the error. The attributes **msg** and **opt** give the error message and related option.

**Example:**

|  |
| --- |
| def main():  host = None  port = None  try:  opts, args = getopt.getopt(sys.argv[1:], "h:p:")  except getopt.GetoptError as err:  print str(err)  usage()  sys.exit(2)  for o, a in opts:  if o == "-h":  host = a  elif o == "-p":  port = a  else: assert False, "unhandled option"  x = PTECMPResolution(host, port)  x.send() |

1. **Python variable types, values and Identifiers**

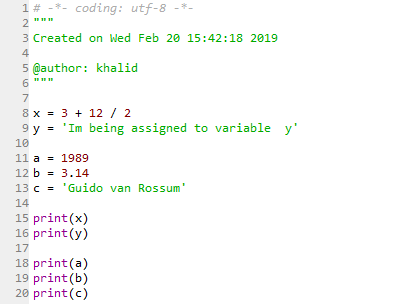
Variables in python are reserved spaces in memory that can hold some value, so whenever a variable is created some space is reserved in memory to hold the value of that variable. Every variable in python is a specific data type which tells the python interpreter how much memory should be reserved for the variable and what type of values the variable can hold.

In python variables don’t need to be declared to assign values to them, so values can be assigned to variable names without explicit declaration of that variable beforehand,the declaration happens when you assign values to them.

* 1. **Assigning values to variables**

Assignments in python creates references not copies.Variables hold references to objects and do not hold the object itself. Names don’t have intrinsic types objects have types, the type of reference is determined by the type of object assigned to the name. Remember the words “Name”, “Variable” and “Identifier” refer to the same thing.

A name is created the first time it appears on the left hand side of an assignment. The code below shows some examples.



This python code when executed gives the following result.



The garbage collector deletes reference after any names that are bound to the reference have passed out of scope. When the reference is deleted the object being referenced is deleted.

In the case where a non existent name(a variable that doesn’t exist yet) is accessed an error is raised. Take a look at the example.



If a name is accessed after assignment it returns the value the name was last assigned.

Take a look at the example given below.



* 1. **Multiple assignment**

Python allows assigning a single value to several names(variables) simultaneously. For example:



Memory is reserved for the integer object with the value 1, and all three variables refer to the same object. Python also also allows assigning multiple objects to multiple variables like the example below.



Here two integer objects are assigned to the names ‘a’ and ‘b’ , a float object is assigned to the name ‘c’ and a string object is assigned to the name ‘d’.

* 1. **Python’s standard data types**

If you have experience in other programming languages you would already know that the value that can be stored in variables are of different types. Having different data types is of course a requirement of programmers in general. Each data type defines the different operations possible and it’s storage mechanism.

Python has five standard data types:

* Integers
* Strings
* List
* Tuple
* Dictionary
  1. **Naming rules**

Names are case sensitive they can contain letters, numbers and underscores but they cant start with a number.

Examples of valid name instances:

Var, VAR, Var1223, V123Bob, c020b

Examples of invalid instances:

1Var, 2a00

Reserved words cant be used as a variable name, so the following cant be used as names.

and, assert, break, class, continue, def, del, elif,

else, except, exec, finally, for, from, global, if,

import, in, is, lambda, not, or, pass, print, raise,

return, try, while

* 1. **Understanding reference semantics**

This may different from the language that you used to work with but in python an assignment operation changes references instead of making a new copy of the object.this can be better understood with an example.



Here x = y doesn't make a copy of the object y references, instead it makes x reference the object(integer 20) that y refers to.This can prove to be very useful but we have to be careful for example take a loo at the code below.



If this confusing it’s Important to first understand what happens when an assignment operation(like x = 3) is performed.

There is a lot that happens in an assignment