G PULLAIAH COLLEGE OF ENGINEERING & TECHNOLOGY
GPCET Pioneering Innovative Education
Accredited by NAAC with 'A' Grade of UGC, Approved by AICTE, New Delhi & Permanently Affiliated to JNTUA, Ananthapuramu
(Recognized by UGC under 2(f) & 12(B) & ISO 9001 : 2008 Certified Institution)
Nandikotkur Road, Venkayapalli, Kurnool-518452

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PYTHON PROGRAMMING (13A05806)

YEAR / SEM: IV / II

PREPARED BY:

Mr. P. Rama Rao

(13A05806) PYTHON PROGRAMMING

UNIT – I:

Introduction: History of Python, Need of Python Programming, Applications Basics of Python Programming Using the REPL(Shell), Running Python Scripts, Variables, Assignment, Keywords, Input-Output, Indentation. Types - Integers, Strings, Booleans;

UNIT – II:

Operators and Expressions: Operators- Arithmetic Operators, Comparison (Relational) Operators, Assignment Operators, Logical Operators, Bitwise Operators, Membership Operators, Identity Operators, Expressions and order of evaluations. **Data Structures** Lists - Operations, Slicing, Methods; Tuples, Sets, Dictionaries, Sequences. Comprehensions.

UNIT – III:

Control Flow - if, if-elif-else, for, while, break, continue, pass

Functions - Defining Functions, Calling Functions, Passing Arguments, Keyword Arguments, Default Arguments, Variable-length arguments, Anonymous Functions, Fruitful Functions(Function Returning Values), Scope of the Variables in a Function - Global and Local Variables.

UNIT – IV:

Modules: Creating modules, import statement, from ..import statement, name spacing,

Python packages, Introduction to PIP, Installing Packages via PIP, Using Python Packages Error and Exceptions: Difference between an error and Exception, Handling Exception, try except block, Raising Exceptions, User Defined Exceptions . **Object Oriented Programming OOP in Python**: Classes, 'self variable', Methods, Constructor Method, Inheritance, Overriding Methods, Data Hiding,

UNIT – V:

Brief Tour of the Standard Library - Operating System Interface - String Pattern Matching, Mathematics, Internet Access, Dates and Times, Data Compression, Multi Threading, GUI Programming, Turtle Graphics. **Testing:** Why testing is required ?, Basic concepts of testing, Unit testing in Python, Writing Test cases, Running Tests.

TEXT BOOKS

1. Python Programming: A Modern Approach, Vamsi Kurama, Pearson
2. Learning Python, Mark Lutz, Orielly

Reference Books:

1. Think Python, Allen Downey, Green Tea Press
2. Core Python Programming, W.Chun, Pearson.
3. Introduction to Python, Kenneth A. Lambert, Cengage

UNIT – I

1. History of Python

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability, and a syntax which allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. CPython, the reference implementation of Python, is open source software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit Python Software Foundation.

Python was conceived in the late 1980s, and its implementation began in December 1989 by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language capable of exception handling and interfacing with the operating system Amoeba. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, benevolent dictator for life (BDFL).

About the origin of Python, Van Rossum wrote in 1996:

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Python 2.0 was released on 16 October 2000 and had many major new features, including a cycle-detecting garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.

Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on 3 December 2008 after a long period of testing. Many of its major features have been back ported to the backwards-compatible Python 2.6.x and 2.7.x version series.

The End Of Life date (EOL, sunset date) for Python 2.7 was initially set at 2015, then postponed to 2020 out of concern that a large body of existing code cannot easily be forward-ported to Python 3. In January 2017 Google announced work on a Python 2.7 to Go transcompiler, which The Register speculated was in response to Python 2.7's planned end-of-life.

Python is a multi-paradigm programming language: object-oriented programming and structured programming are fully supported, and many language features support functional programming

and aspect-oriented programming. Many other paradigms are supported via extensions, including design by contract and logic programming.

Python uses dynamic typing and a mix of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.

The design of Python offers some support for functional programming in the Lisp tradition. The language has map(), reduce() and filter() functions; list comprehensions, dictionaries, and sets; and generator expressions. The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and Standard ML.

Rather than requiring all desired functionality to be built into the language's core, Python was designed to be highly extensible. Python can also be embedded in existing applications that need a programmable interface. This design of a small core language with a large standard library and an easily extensible interpreter was intended by Van Rossum from the start because of his frustrations with ABC, which espoused the opposite mindset.

Python's developers strive to avoid premature optimization, and moreover, reject patches to non-critical parts of CPython that would offer a marginal increase in speed at the cost of clarity.[46] When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C, or try using PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

An important goal of Python's developers is making it fun to use. This is reflected in the origin of the name, which comes from Monty Python, and in an occasionally playful approach to tutorials and reference materials, such as using examples that refer to spam and eggs instead of the standard foo and bar.

A common neologism in the Python community is pythonic, which can have a wide range of meanings related to program style. To say that code is pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language, that it conforms with Python's minimalist philosophy and emphasis on readability. In contrast, code that is difficult to understand or reads like a rough transcription from another programming language is called unpythonic.

Users and admirers of Python, especially those considered knowledgeable or experienced, are often referred to as Pythonists, Pythonistas, and Pythoneers.

- **Paradigm multi-paradigm**: object-oriented, imperative, functional, procedural, reflective
- **Designed by:** Guido van Rossum
- **Developer:** Python Software Foundation
- **First appeared**: 20 February 1991; 25 years ago[1]
- **Stable release**
  3.6.0 / 23 December 2016; 55 days ago[2]
  2.7.13 / 17 December 2016; 61 days ago[3]
- **OS**: Cross-platform

- **License:** Python Software Foundation License
- **Filename extensions:** .py, .pyc, .pyd, .pyo (prior to 3.5).pyw, .pyz (since 3.5)
- **Major implementations**
  CPython, IronPython, Jython, MicroPython, PyPy
- **Dialects**
  Cython, RPython, Stackless Python
- **Influenced by**
  ABC,ALGOL 68, C,C++,Dylan,Haskell,Icon,Java,Lisp,Modula3, Perl

2. Need of Python Programming

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the scripting world. Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and function programming.

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically *one-third to* less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed. *Program portability* Most Python programs run unchanged on *all major computer platforms*. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web based systems, and more. Even operating system interfaces, including program launches and directory processing, are as portable in Python as they can possibly be.

Support libraries

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both home grown libraries and a vast collection of third-party application support software. Python's *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling). The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

Component integration

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

Enjoyment

Because of Python's ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this may be an intangible benefit, its effect on productivity is an important asset. Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

Software Quality

By design, Python implements a deliberately simple and readable syntax and a highly coherent programming model. As a slogan at a past Python conference attests, the net result is that Python seems to "fit your brain"—that is, features of the language interact Python is perhaps best described as an object-oriented scripting language: its design mixes software engineering features of traditional languages with the usability of scripting languages. But some of Python's best assets tell a more complete story. in consistent and limited ways and follow naturally from a small set of core concepts. This makes the language easier to learn, understand, and remember. In practice, Python programmers do not need to constantly refer to manuals when reading or writing code; it's a consistently designed system that many find yields surprisingly uniform code. By philosophy, Python adopts a somewhat minimalist approach. This means that although there are usually multiple ways to accomplish a coding task, there is usually just one obvious way, a few less obvious alternatives, and a small set of coherent interactions everywhere in the language. Moreover, Python doesn't make arbitrary decisions for you; when interactions are ambiguous, explicit intervention is preferred over "magic." In the Python way of thinking, explicit is better than implicit, and simple is better than complex.Beyond such design themes, Python includes tools such as modules and OOP that naturally promote code reusability. And because Python is focused on quality, so too, naturally, are Python programmers.

Developer Productivity

During the great Internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the Internet evolved. In later eras of layoffs and economic recession, the picture shifted. Programming staffs were often asked to accomplish the same tasks with even fewer people. In both of these scenarios, Python has shined as a tool that allows programmers to get more done with less effort. It is deliberately optimized for speed of development—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools. The net effect is that Python typically boosts developer productivity many times beyond the levels supported by traditional languages. That's good news in both boom and bust times, and everywhere the software industry goes in between.

It's Object-Oriented

Python is an object-oriented language, from the ground up. Its class model supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet in the context of Python's dynamic typing, object-oriented programming (OOP) is remarkably easy to apply. In fact, if you don't understand these terms, you'll find they are much easier to learn with Python than with just about any other OOP language available. Besides serving as a powerful code structuring and reuse device, Python's OOP nature makes it ideal as a scripting tool for object-oriented systems languages such as C++ and Java. For example, with the appropriate glue code, Python programs can subclass (specialize) classes implemented in C++ or Java. Of equal significance, OOP is an option in Python; you can go far without having to become an object guru all at once.

It's Free

Python is freeware—something which has lately been come to be called *open source software*. As with Tcl and Perl, you can get the entire system for free over the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python, if you're so inclined. But don't get the wrong idea: "free" doesn't mean "unsupported." On the contrary, the Python online community responds to user queries with a speed that most commercial software vendors would do well to notice. Moreover, because Python comes with complete source code, it empowers developers and creates a large team of implementation experts. Although studying or changing a programming language's implementation isn't everyone's idea of fun, it's comforting to know that it's available as a final resort and ultimate documentation source.

It's Portable

Python is written in portable ANSI C, and compiles and runs on virtually every major platform in use today. For example, it runs on Unix systems, Linux, MS-DOS, MS-Windows (95, 98, NT), Macintosh, Amiga, Be-OS, OS/2, VMS, QNX, and more. Further, Python programs are automatically compiled to portable *bytecode*, which runs the same on any platform with a compatible version of Python installed (more on this in the section "It's easy to use"). What that means is that Python programs that use the core language run the same on Unix, MS-Windows, and any other system with a Python interpreter. Most Python ports also contain platform-specific extensions (e.g., COM support on MS-Windows), but the core Python language and libraries work the same everywhere. Python also includes a standard interface to the Tk GUI system called Tkinter, which is portable to the X Window System, MS Windows, and the Macintosh, and now provides a native look-and-feel on each platform. By using Python's Tkinter API, Python programs can implement full-featured graphical user interfaces that run on all major GUI platforms without program changes.

It's Powerful

From a features perspective, Python is something of a hybrid. Its tool set places it between traditional scripting languages (such as Tcl, Scheme, and Perl), and systems languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced programming tools typically found in systems development languages. Unlike some scripting languages, this combination makes Python useful for substantial development projects. Some of the things we'll find in Python's high-level toolbox: **Dynamic typing** Python keeps track of the kinds of objects your program uses when it runs; it doesn't require complicated type and size declarations in your code.

Built-in object types Python provides commonly used data structures such as lists, dictionaries, and strings, as an intrinsic part of the language; as we'll see, they're both flexible and easy to use.

Built-in tools To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

Library utilities For more specific tasks, Python also comes with a large collection of precoded library tools that support everything from regular-expression matching to networking to object persistence.

Third-party utilities Because Python is freeware, it encourages developers to contribute precoded tools that support tasks beyond Python's built-ins; you'll find free support for COM, imaging, CORBA ORBs, XML, and much more.

Automatic memory management
Python automatically allocates and reclaims ("garbage collects") objects when no longer used, and most grow and shrink on demand; Python, not you, keeps track of low-level memory details.

Programming-in-the-large support Finally, for building larger systems, Python includes tools such as modules, classes, and exceptions; they allow you to organize systems into components, do OOP, and handle events gracefully.

Despite the array of tools in Python, it retains a remarkably simple syntax and design. As we'll see, the result is a powerful programming tool, which retains the usability of a scripting language.

It's Mixable

Python programs can be easily "glued" to components written in other languages. In technical terms, by employing the Python/C integration APIs, Python programs can be both extended by (called to) components written in C or C++, and embedded in (called by) C or C++ programs.
That means you can add functionality to the Python system as needed and use Python programs within other environments or systems. Although we won't talk much about Python/C integration, it's a major feature of the language and one reason Python is usually called a scripting language. By mixing Python with components written in a compiled language such as C or C++, it becomes an easy-to-use frontend language and customization tool. It also makes Python good at rapid prototyping: systems may be implemented in Python first to leverage its speed of development, and later moved to C for delivery, one piece at a time, according to performance requirements. Speaking of glue, the PythonWin port of Python for MS-Windows platforms also lets Python programs talk to other components written for the COM API, allowing Python to be used as a more powerful alternative to Visual Basic. And a new alternative implementation of Python, called JPython, lets Python programs communicate with Java programs, making Python an ideal tool for scripting Java-based web applications.

It's Easy to Use

For many, Python's combination of rapid turnaround and language simplicity make programming more fun than work. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps (as when using languages such as C or C++). As with other interpreted languages, Python executes programs immediately, which makes for both an interactive programming experience and rapid turnaround after program changes. Strictly speaking, Python programs are compiled (translated) to an intermediate form called *bytecode*, which is then run by the interpreter. But because the compile step is automatic and hidden to programmers, Python achieves the development speed of an interpreter without the performance loss inherent in purely interpreted languages. Of course, development cycle turnaround is only one aspect of Python's ease of use. It also provides a deliberately simple syntax and powerful high-level built-in tools. Python has been called "executable pseudocode": because it eliminates much of the complexity in other tools, you'll find that Python programs are often a fraction of the size of equivalent programs in languages such as C, C++, and Java.

It's Easy to Learn

This brings us to the topic of this book: compared to other programming languages, the core Python language is amazingly easy to learn. In fact In fact, you can expect to be coding significant Python programs in a matter of days (and perhaps in just hours, if you're already an experienced programmer). That's good news both for professional developers seeking to learn the language to use on the job, as well as for end users of systems that expose a Python layer for customization or control.

Python on the Job

Besides being a well-designed programming language, Python is also useful for accomplishing real-world tasks—the sorts of things developers do day in and day out. It's commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs. Some of Python's major roles help define what it is.

System Utilities

Python's built-in interfaces to operating-system services make it ideal for writing portable, maintainable system-administration tools (sometimes called shell scripts). Python comes with POSIX bindings and support for the usual OS tools: environment variables, files, sockets, pipes, processes, threads, regular expressions, and so on. **GUIs** Python's simplicity and rapid turnaround also make it a good match for GUI programming. As previously mentioned, it comes with a standard object-oriented interface to the Tk GUI API called Tkinter, which allows Python programs to implement portable GUIs with native look and feel. If portability isn't a priority, you can also use MFC classes to build GUIs with the PythonWin port for MS Windows, X Window System interfaces on Unix, Mac toolbox bindings on the Macintosh, and KDE and GNOME interfaces for Linux. For applications that run in web browsers, JPython provides another GUI option.

Component Integration

Python's ability to be extended by and embedded in C and C++ systems makes it useful as a glue language, for scripting the behavior of other systems and components. For instance, by integrating a C library into Python, Python can test and launch its components. And by embedding Python in a product, it can code onsite customizations without having to recompile the entire product (or ship its source code to your customers). Python's COM support on MS-Windows and the JPython system provide alternative ways to script applications.

Rapid Prototyping

To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially and then move components to a compiled language such as C or C++ for delivery. Unlike some prototyping tools, Python doesn't require a complete rewrite once the prototype has solidified; parts of the system that don't require the efficiency of a language such as C++ can remain coded in Python for ease of maintenance and use.

Internet Scripting

Python comes with standard Internet utility modules that allow Python programs to communicate over sockets, extract form information sent to a server-side CGI script, parse HTML, transfer files by FTP, process XML files, and much more. There are also a number of peripheral tools for doing Internet programming in Python. For instance, the HTMLGen and pythondoc systems generate HTML files from Python class-based descriptions, and the JPython system mentioned above provides for seamless Python/Java integration.

Numeric Programming The NumPy numeric programming extension for Python includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy turns Python into a sophisticated yet easy-to-use numeric programming tool.

Database Programming

Python's standard pickle module provides a simple object-persistence system: it allows programs to easily save and restore entire Python objects to files. For more traditional database demands, there are Python interfaces to Sybase, Oracle, Informix, ODBC, and more. There is even a portable SQL database API for Python that runs the same on a variety of underlying database systems, and a system named *gadfly* that implements an SQL database for Python programs.

And More: Image Processing, AI, Distributed Objects, Etc.

Python is commonly applied in more domains than can be mentioned here. But in general, many are just instances of Python's component integration role in action. By adding Python as a frontend to libraries of components written in a compiled language such as C, Python becomes useful for scripting in a variety of domains. For instance, image processing for Python is implemented as a set of library components implemented in a compiled language such as C, along with a Python frontend layer on top used to configure and launch the compiled components. The easy-to-use Python layer complements the efficiency of the underlying compiled-language components. Since the majority of the "programming" in such a system is done in the Python layer, most users need never deal with the complexity of the optimized components (and can get by with the core language covered in this text). ***Python in* Commercial Products**

From a more concrete perspective, Python is also being applied in real revenue-generating products, by real companies. For instance, here is a partial list of current Python users:

- Red Hat uses Python in its Linux install tools.
- Microsoft has shipped a product partially written in Python.
- Infoseek uses Python as an implementation and end-user customization language in web search products.
- Yahoo! uses Python in a variety of its Internet services.
- NASA uses Python for mission-control-system implementation.
- Lawrence Livermore Labs uses Python for a variety of numeric programming tasks.
- Industrial Light and Magic and others use Python to produce commercial-grade animation.

There are even more exciting applications of Python we'd like to mention here, but alas, some companies prefer not to make their use of Python known because they consider it to be a competitive advantage.

Who Uses Python Today?

1. *Google* makes extensive use of Python in its web search systems.
2. The popular *YouTube* video sharing service is largely written in Python.
3. The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
4. The *Raspberry Pi* single-board computer promotes Python as its educational language.
5. The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.
6. Google's *App Engine* web development framework uses Python as an application language.
7. *Maya*, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
8. *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm*, and *IBM* use Python for hardware testing.
9. *NASA*, *Los Alamos*, *Fermilab*, *JPL*, and others use Python for scientific programming tasks.

Some of Python's most common applications today:

1. Systems Programming
2. GUIs
3. Internet Scripting

4. Component Integration
5. Database Programming
6. Rapid Prototyping
7. Numeric and Scientific Programming

What Are Python's Technical Strengths?

1. It's Object-Oriented and Functional
2. It's Free
3. It's Portable
4. It's Powerful
5. It's Mixable
6. It's Relatively Easy to Use
7. It's Relatively Easy to Learn

3. Running Python Scripts

Introducing the Python Interpreter

An interpreter is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine. When the Python package is installed on your machine, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or something else. Whatever form it takes, the Python code you write must always be run by this interpreter. And to enable that, you must install a Python interpreter on your computer.

• Windows users fetch and run a self-installing executable file that puts Python on their machines. Simply double-click and say Yes or Next at all prompts.
• Linux and Mac OS X users probably already have a usable Python preinstalled on their computers—it's a standard component on these platforms today.
• Some Linux and Mac OS X users (and most Unix users) compile Python from its full source code distribution package.

Program Execution

What it means to write and run a Python script depends on whether you look at these tasks as a programmer, or as a Python interpreter. Both views offer important perspectives on Python programming.

The Programmer's View

In its simplest form, a Python program is just a text file containing Python statements. For example, the following file, named *script0.py*, is one of the simplest Python scripts I could dream up, but it passes for a fully functional Python program:

```
print('hello world')
print(2 ** 100)
```

This file contains two Python print statements, which simply print a string (the text in quotes) and a numeric expression result (2 to the power 100) to the output stream.

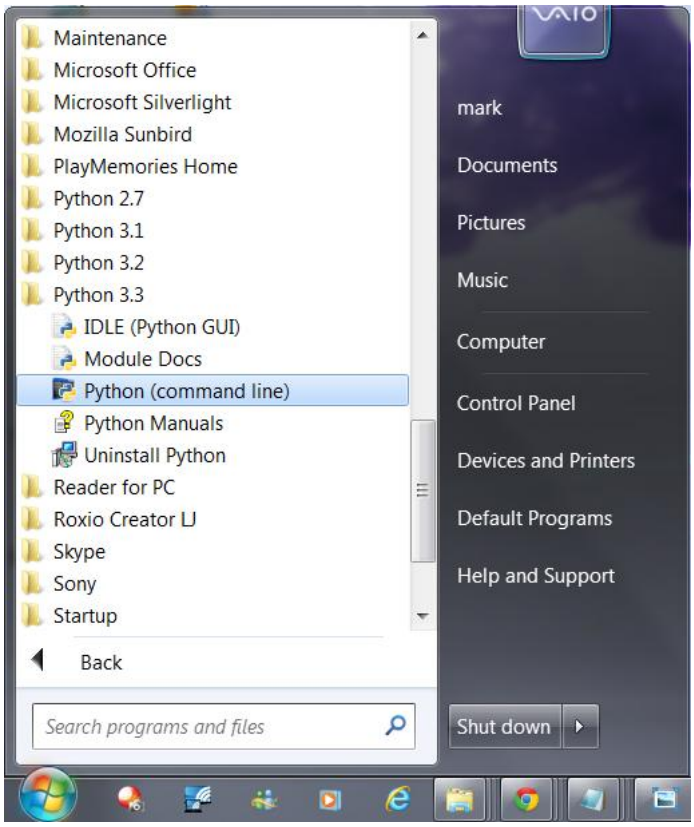


Figure : IDLE in Startup menu

You can create such a file of statements with any text editor you like. By convention, Python program files are given names that end in *.py*; technically, this naming scheme is required only for files that are imported"—a term clarified in the next chapter—but most Python files have *.py* names for consistency.

After you've typed these statements into a text file, you must tell Python to *execute* the file—which simply means to run all the statements in the file from top to bottom, one after another. You can launch Python program files by shell command lines, by clicking their icons, from within IDEs, and with other standard techniques. If all goes well, when you execute the file, you'll see the results of the two print statements show up somewhere on your computer—by default, usually in the same window you were in when you ran the program:

```
hello world
126765060022822940149670320537 6
```

For example, here's what happened when I ran this script from a Command Prompt window's command line on a Windows laptop, to make sure it didn't have any silly typos:

```
C:\code> python script0.py
hello world
126765060022822940149670320537 6
```

Python's View

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts crunching away. Specifically, it's first compiled to something called "byte code" and then routed to something called a "virtual machine."

Byte code compilation

Python first compiles your *source code* (the statements in your file) into a format known as *byte code*. Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution —byte code can be run much more quickly than the original source code statements in your text file.

The Python Virtual Machine (PVM)

Once your program has been compiled to byte code (or the byte code has been loaded from existing *.pyc* files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM).

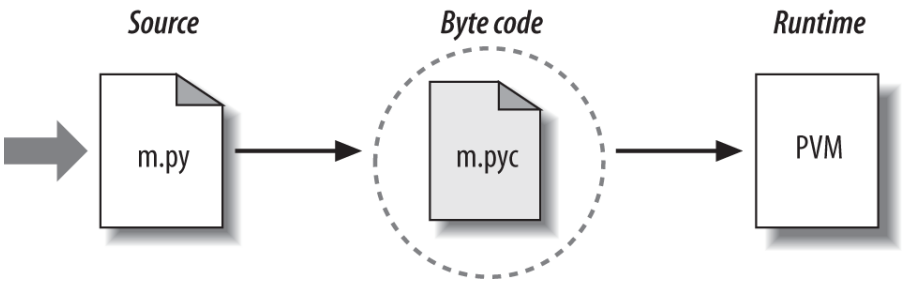


Figure: Python's traditional runtime execution model

Source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

The Interactive Prompt

Starting an Interactive Session

Perhaps the simplest way to run Python programs is to type them at Python's interactive command line, sometimes called the *interactive prompt*. There are a variety of ways to start this command line: in an IDE, from a system console, and so on. Assuming the interpreter is installed as an executable program on your system, the most platform neutral way to start an interactive interpreter session is usually just to type **python** at your operating system's prompt, without any arguments. For example:

```
% python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Typing the word "python" at your system shell prompt like this begins an interactive Python session; the "%" character at the start of this listing stands for a generic system prompt in this

book—it's not input that you type yourself. On Windows, a *Ctrl-Z* gets you out of this session; on Unix, try *Ctrl-D* instead. The notion of a *system shell prompt* is generic, but exactly how you access it varies by platform:

1. On *Windows*, you can type **python** in a DOS console window—a program named cmd.exe and usually known as *Command Prompt*. For more details on starting this program,
2. On *Linux* (and other Unixes), you might type this command in a shell or terminal window

The System Path

When we typed **python** in the last section to start an interactive session, we relied on the fact that the system located the Python program for us on its program search path. Depending on your Python version and platform, if you have not set your system's PATH environment variable to include Python's install directory, you may need to replace the word "python" with the full path to the Python executable on your machine. On Unix, Linux, and similar, something like **/usr/local/bin/python** or **/usr/bin/python3** will often suffice. On Windows, try typing **C:\Python33\python** (for version 3.3):

```
c:\code> c:\python33\python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Alternatively, you can run a "cd" change-directory command to go to Python's install directory before typing **python**—try the **cd c:\python33** command on Windows, for example:

```
c:\code> cd c:\python33
c:\Python33> python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

In Linux:

Open a terminal and type
export PATH="$PATH:/usr/local/bin/python"
and press Enter

In Windows:

At the command prompt, type
path %path%;C:\Python
and press Enter.

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

PYTHONPATH: Tells the Python interpreter where to locate the module files imported into a program. PYTHONPATH is sometimes preset by the Python installer.

The IDLE User Interface

So far, we've seen how to run Python code with the interactive prompt, system command lines, Unix-style scripts, icon clicks, module imports, and exec calls. If you're looking for something a bit more visual, *IDLE* provides a graphical user interface for doing Python development, and it's a standard and free part of the Python system. IDLE is usually referred to as an *integrated development environment* (IDE), because it binds together various development tasks into a single view. In short, IDLE is a desktop GUI that lets you edit, run, browse, and debug Python programs, all from a single interface. It runs portably on most Python platforms, including Microsoft

Windows, X Windows (for Linux, Unix, and Unix-like platforms), and the Mac OS (both Classic and OS X). For many, IDLE represents an easy-to-use alternative to typing command lines, a less problem-prone alternative to clicking on icons, and a great way for newcomers to get started editing and running code.

IDLE Startup Details

Most readers should be able to use IDLE immediately, as it is a standard component on Mac OS X and most Linux installations today, and is installed automatically with standard Python on Windows. Because platforms specifics vary, though, I need to give a few pointers before we open the GUI. Technically, IDLE is a Python program that uses the standard library's tkinter GUI toolkit (named Tkinter in Python 2.X) to build its windows. This makes IDLE portable —it works the same on all major desktop platforms—but it also means that you'll need to have tkinter support in your Python to use IDLE. This support is standard on Windows, Macs, and Linux, but it comes with a few caveats on some systems, and startup can vary per platform. Here are a few platform-specific tips: On *Windows 7* and earlier, IDLE is easy to start—it's always present after a Python install, and has an entry in the Start button menu for Python in Windows 7 and earlier. You can also select it by right-clicking on a Python program icon, and launch it by clicking on the icon for the files *idle.pyw* or *idle.py* located in the *idlelib* subdirectory of Python's *Lib* directory. In this mode, IDLE is a clickable Python script that lives in *C:\Python33\Lib\idlelib*, *C:\Python27\Lib\idlelib*, or similar, which you can drag out to a shortcut for one click access if desired.

- On *Windows 8*, look for IDLE in your Start tiles, by a search for "idle," by browsing your "All apps" Start screen display, or by using File Explorer to find the *idle.py* file mentioned earlier. You may want a shortcut here, as you have no Start button menu in desktop mode
- On *Mac OS X* everything required for IDLE is present as standard components in your operating system. IDLE should be available to launch in *Applications* under the *MacPython* (or *Python N.M*) program folder. One note here: some OS X versions may require installing updated tkinter support due to subtle version dependencies .
- On *Linux* IDLE is also usually present as a standard component today. It might take the form of an *idle* executable or script in your path; type this in a shell to check. On some machines, it may require an install and on others you may need to launch IDLE's top-level script from a command line or icon click: run the file *idle.py* located in the *idlelib* subdirectory of Python's */usr/lib* directory (run a find for the exact location).

IDLE Basic Usage

Let's jump into an example. Figure shows the scene after you start IDLE on Windows. The Python shell window that opens initially is the main window, which runs an interactive session (notice the >>> prompt). This works like all interactive sessions —code you type here is run immediately after you type it—and serves as a testing and experimenting tool.

IDLE uses familiar menus with keyboard shortcuts for most of its operations. *To make a new script file* under IDLE, use File→New: in the main shell window, select the File pull-down menu, and pick New to open a new text edit window where you can type, save, and run your file's code. Use File→Open... instead to open a new text edit window displaying an existing file's code to edit and run. Although it may not show up fully in this book's graphics, IDLE uses syntax-directed *colorization* for the code typed in both the main window and all text edit windows— keywords are one color, literals are another, and so on. This helps give you a better picture of the components in your code (and can even help you spot mistakes—run on strings are all one color, for example).

To run a file of code that you are editing in IDLE, use Run→Run Module in that file's text edit window. That is, select the file's text edit window, open that window's *Run* pull-down menu, and choose the *Run Module* option listed there (or use the equivalent keyboard shortcut, given in the menu). Python will let you know that you need to save your file first if you've changed it since it was opened or last saved and forgot to save your changes—a common mistake when you're knee-deep in coding. When run this way, the output of your script and any error messages it may generate show up back in the main interactive window (the Python shell window). In Figure , for example, the three lines after the "RESTART" line near the middle of the window reflect an execution of our *script1.py* file opened in a separate edit window. The "RESTART" message tells us that the user-code process was restarted to run the edited script and serves to separate script output .

IDLE Usability Features

Like most GUIs, the best way to learn IDLE may be to test-drive it for yourself, but some key usage points seem to be less than obvious. For example, if you want to *repeat prior commands* in IDLE's main interactive window, you can use the *Alt-P* key combination to scroll backward through the command history, and *Alt-N* to scroll forward (on some Macs, try Ctrl-P and Ctrl-N instead). Your prior commands will be recalled and displayed, and may be edited and rerun. You can also recall commands by positioning the *cursor* on them and clicking and pressing Enter to insert their text at the input prompt, or using standard cut-and-paste operations, though these techniques tend to involve more steps (and can sometimes be triggered accidentally).

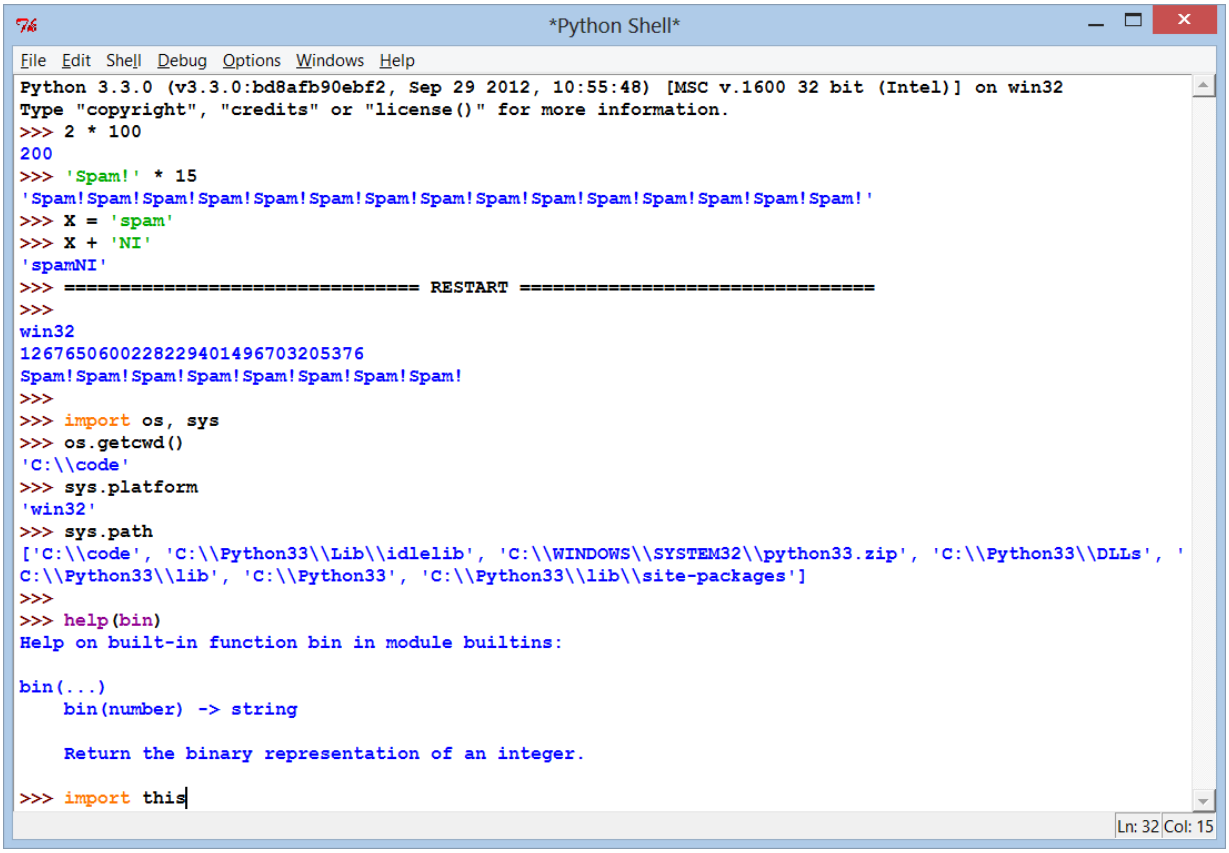


Figure :The main Python shell window of the IDLE development GUI

Outside IDLE, you may be able to recall commands in an interactive session with the arrow keys on Windows. Besides command history and syntax *colorization*, IDLE has additional usability features such as:

1. *Auto-indent* and unindent for Python code in the editor (Backspace goes back one level)
2. Word *auto-completion* while typing, invoked by a Tab press
3. Balloon help pop ups for a *function call* when you type its opening "("
4. Pop-up selection lists of *object attributes* when you type a "." after an object's name
5. and either pause or press Tab

Some of these may not work on every platform, and some can be configured or disabled if you find that their defaults get in the way of your personal coding style.

Apart from IDLE, here are some of Python's most commonly used IDEs:

1. *Eclipse and PyDev*
2. *Komodo*
3. *NetBeans IDE for Python*
4. *PythonWin*
5. *Wing, Visual Studio, and others*

4. Variables and Assignment

Variables are nothing but reserved memory locations to store values. This means when you create a variable, you reserve some space in memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Rules for variables in Python are the same as they are in most other high-level languages inspired by (or more likely, written in) C.
They are simply identifier names with an alphabetic first character "alphabetic" meaning upper-or lowercase letters, including the underscore (_). Any additional characters may be alphanumeric or underscore. Python is case-sensitive, meaning that the identifier "cAsE" is different from "CaSe."

```
>>> counter = 0
>>> miles = 1000.0
>>> name = 'Bob'
>>> counter = counter + 1
>>> kilometers = 1.609 * miles
>>> print '%f miles is the same as %f km' % (miles, kilometers)
1000.000000 miles is the same as 1609.000000 km
```

We have presented five examples of variable assignment. The first is an integer assignment followed by one each for floating point numbers, one for strings, an increment statement for integers, and finally, a floating point operation and assignment.
Python also supports augmented assignment, statements that both refer to and assign values to variables. You can take the following expression ...

```
n = n * 10
```

...and use this shortcut instead:

```
n *= 10
```

Python does not support increment and decrement operators like the ones in C: n++ or --n. Because ++ and -- are also unary operators, Python will interpret --n as -(-n) == n, and the same is

true for ++n. Python allows you to assign a single value to several variables simultaneously. For example:
a = b = c = 1
Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example:
a, b, c = 1, 2, "john"
Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

5. KEYWORDS

The following list shows the Python keywords. These are reserved words and you cannot use them 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	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

6. INPUT

To get input from the user you can use the input function. When the input function is called the program stops running the program, prompts the user to enter something at the keyboard by printing a string called the prompt to the screen, and then waits for the user to press the Enter key. The user types a string of characters and presses enter. Then the input function returns that string and Python continues running the program by executing the next statement after the input statement.

Example : Consider this short program.

```
name = input("Please enter your name:")

print("The name you entered was", name)
```

The input function prints the prompt “Please enter your name:” to the screen and waits for the user to enter input in the Python Shell window. The program does not continue executing until you have provided the input requested. When the user enters some characters and presses enter, Python

takes what they typed before pressing enter and stores it in the variable called name in this case. The type of value read by Python is always a string. If we want to convert it to an integer or some other type of value, then we need to use a conversion operator. For instance, if we want to get an int from the user, we must use the int conversion operator.

Example:

```
age = int(input("Please enter your age:"))

olderAge = age + 1

print("Next year you will be", olderAge)
```

7. OUTPUT(print):

Prints the values to a output window.
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
or
print(value1,value2,…)
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
When printing, we may print as many items as we like on one line by separating each item by a comma. Each time a comma appears between items in a print statement, a space appears in the output.
Example :Here is some code that prints a few values to the screen.

```
name = "Sophus"
print(name,"how are you doing?")
print("I hope that,", name, "is feeling well today.")
```

The output from this is:
Sophus how are your doing?
I hope that Sophus is feeling well today.

To print the contents of variables without spaces appearing between the items, the variables must be converted to strings and string concatenation can be used. The + operator adds numbers together, but it also concatenates strings. For the correct + operator to be called, each item must first be converted to a string before concatenation can be performed.
Example:

Create a file in IDLE and save it as add.py

```
x = input("Enter an integer: ")
y = input("Enter another integer: ")
z=int(x)+int(y)
print("sum of ", x, " and ", y, " is ", z, ".", sep=" ")
```

Output
Enter an integer: 23
Enter another integer: 12
sum of 23 and 12 is 35 .

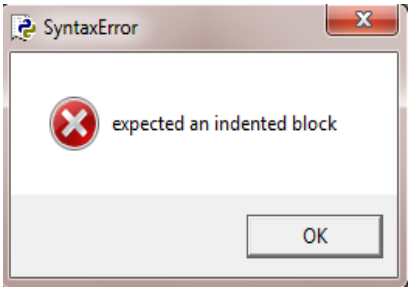
8. Indentation

Code blocks are identified by indentation rather than using symbols like curly braces. Without extra symbols, programs are easier to read. Also, indentation clearly identifies which block of code a statement belongs to. Of course, code blocks can consist of single statements, too. When one is new to Python, indentation may comes as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy to read.

Python does not support braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. All the continuous lines indented with same number of spaces would form a block. Python strictly follow indentation rules to indicate the blocks.

Example 1:

```
if True:
    print("True")
else:
    print("False")
```



Example:

```
if True:
print("True")
else:
print("False")
```

9. DATA TYPES

Python data types are:

1. Numbers
2. Booleans
3. Strings

9. 1 NUMBERS

Python has 4 built-in numeric data types:

1. Integers
2. Long integers
3. Floating point numbers
4. Imaginary numbers

Constant	**Interpretation**
1234, -24, 0	Normal intergers (C longs)
9999999999999L	Long integers (unlimited size)
1.23, 3.14e-10, 4E210, 4.0e+210	Floating-point (C doubles)

0177, 0x9ff	Octal and hex constants
3+4j, 3.0+4.0j, 3J	Complex number constants

By and large, Python's numeric types are straightforward, but a few are worth highlighting here:

Integer and floating-point constants

Integers are written as a string of decimal digits. Floating-point numbers have an embedded decimal point, and/or an optional signed exponent introduced by an e or E. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when it's used in an expression. The rules for writing floating-point numbers are the same as with C.

Numeric precision Plain Python integers (row 1) are implemented as C longs internally (i.e., at least 32 bits), and Python floating-point numbers are implemented as C doubles; Python numbers get as much precision as the C compiler used to build the Python interpreter gives to longs and doubles. On the other hand, if an integer constant ends with an l or L, it becomes a Python long integer (not to be confused with a C long) and can grow as large as needed.

Hexadecimal and octal constants

The rules for writing hexadecimal (base 16) and octal (base 8) integers are the same as in C: octal constants start with a leading zero (0), and hexadecimals start with a leading 0x or 0X. Notice that this means you can't write normal base-ten integers with a leading zero (e.g., 01); Python interprets them as octal constants, which usually don't work as you'd expect!

Long integers

Now for something more exotic: here's a look at long integers in action. When an integer constant ends with an L (or lowercase l), Python creates a long integer, which can be arbitrarily big:

```
>>> 99999999999999999999999999999 + 1
OverflowError: integer literal too large
>>> 99999999999999999999999999999L + 1
100000000000000000000000000000L
```

Here, the first expression fails and raises an error, because normal integers can't accommodate such a large number. On the other hand, the second works fine, because we tell Python to generate a long integer object instead. Long integers are a convenient tool. In fact, you can use them to count the national debt in pennies, if you are so inclined. But because Python must do extra work to support their extended precision, long integer math is usually much slower than normal integer math. If you need the precision, it's built in for you to use.

Complex numbers

Python complex constants are written as real-part + imaginary-part, and terminated with a j or J. Internally, they are implemented as a pair of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers are a recent addition to Python. Complex numbers are represented as two floating-point numbers—the *real* and *imaginary* parts—and are coded by adding a j or J suffix to the imaginary part. We can also write complex numbers with a nonzero real part by adding the two parts with a +. For example, the complex number with a real part of 2 and an imaginary part of -3 is written: 2 + -3j. Some examples of complex math at work:

```
>>> 1j * 1J
(-1+0j)
```

```
>>> 2 + 1j * 3
(2+3j)
>>> (2+1j)*3
(6+3j)
```

Attribute	*Description*
num.real	Real component of complex number *num*
num.imag	Imaginary component of complex number *num*
num.conjugate()	Returns complex conjugate of *num*

9. 2 BOOLEAN

Here are some of the major concepts surrounding Boolean types:
They have a constant value of either true or False.
Booleans are subclassed from integers but cannot themselves be further derived.
Recall that Python objects typically have a Boolean False value for any numeric zero or empty set.
Also, if used in an arithmetic context, Boolean values TRue and False will take on their numeric equivalents of 1 and 0, respectively.
Most of the standard library and built-in Boolean functions that previously returned integers will now return Booleans.
All Python objects have an inherent true or False value
Here are some examples using Boolean values:

```
# intro
>>> bool(1)
True
>>> bool(True)
True
>>> bool(0)
False
>>> bool('1')
True
>>> bool('0')
True
>>> bool([])
False
>>> bool ( (1,) )
True
# using Booleans numerically
>>> foo = 42
>>> bar = foo < 100
>>> bar
True
>>> print bar + 100
101
>>> print '%s' % bar
True
>>> print '%d' % bar
1
```

9. 3 STRINGS

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. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

```
>>> c='aaaaaaaa bbbb cccc dddd eee ffff'
>>> c
'aaaaaaaa bbbb cccc dddd eee ffff‘
>>> st2="aaaaaaaa bbbb ccccc dddd eee ffff"
>>> st2
'aaaaaaaa bbbb ccccc dddd eee ffff‘
>>> st3="""aaaaa bbbbb
ccccc ddddd
eeeee fffff"""“
>>> st3
'aaaaa bbbbb\nccccc ddddd\neeeee fffff'
>>>
str = 'Hello World!'
print(str)
print(str[0])
print(str[2:5])
print(str[2:])
print(str * 2)
print(str + "TEST")
```

Output:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string all together.

For example:

```
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result:

```
Updated String :- Hello Python
```

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation. An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\e	0x1b	Escape
\f	0x0c	Formfeed
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then:

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give - HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

String Formatting Operator

One of Python's coolest features is the string format operator **%**. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example:

print "My name is %s and weight is %d kg!" % ('Zara', 21)

When the above code is executed, it produces the following result:

My name is Zara and weight is 21 kg!

Here is the list of complete set of symbols which can be used along with %:

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Other supported symbols and functionality are listed in the following table:

Symbol	Functionality
*	argument specifies width or precision
-	left justification
+	display the sign
<sp>	leave a blank space before a positive number
#	add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0	pad from left with zeros (instead of spaces)
%	'%%' leaves you with a single literal '%'
(var)	mapping variable (dictionary arguments)

String Methods:

1. capitalize() - Capitalizes first letter of string.

```
>>>st="hello to all“

>>> st.capitalize()

'Hello to all'
```

2. **endswith()-**

endswith(suffix, beg=0, end=len(string))

```
str = "this is string example....wow!!!";

suffix = "wow!!!";
```

```
print str.endswith(suffix);
print str.endswith(suffix,20);
suffix = "is";
print str.endswith(suffix, 2, 4);
print str.endswith(suffix, 2, 6);
```

Output:True

True

True

False

3. **find()**

find(str, beg=0 end=len(string))

```
str1 = "this is string example....wow!!!";
str2 = "exam";
print str1.find(str2);
print str1.find(str2, 10);
print str1.find(str2, 40);
```

Output: 15 15 -1

4. isa1num ()-It checks whether the string consists of alphanumeric characters.

```
str = "this2009"; # No space in this string
print str.isalnum();
str = "this is string example....wow!!!";
print str.isalnum();
```

Output: True False

5. isdigit ()- Checks whether the string consists of digits only.

```
str = "123456";
print str.isdigit();
str = "this is string example....wow!!!";
print str.isdigit();
```

Output:

True

False

6. **islower ()-** Checks whether all the case-based characters (letters) of the string are lowercase.

```
str = "THIS is string example....wow!!!";
print str.islower();
str = "this is string example....wow!!!";
print str.islower();
```

Output:

False

True

7. isspace()- Checks whether the string consists of whitespace.

```
str = " ";
print str.isspace();
str = "This is string example....wow!!!";
print str.isspace();
```

Output:

True

False

8. istitle ()

```
str = "This Is String Example...Wow!!!";
print str.istitle();
str = "This is string example....wow!!!";
print str.istitle();
```

Output

True

False

9. isupper() - Checks whether all the case-based characters (letters) of the string are uppercase.

```
str = "THIS IS STRING EXAMPLE....WOW!!!";

print str.isupper();

str = "THIS is string example....wow!!!";

print str.isupper();
```

Output: True False

10. join()- Returns a string in which the string elements of sequence have been joined by *str separator.*

```
str = "-";

seq = ("a", "b", "c");

print str.join( seq );
```

Output:

a-b-c

11. len ()- Returns the length of the string.

```
str = "this is string example....wow!!!";

print "Length of the string: ", len(str);
```

Output:

Length of the string: 32

12 lower ()-Returns a copy of the string in which all case-based characters have been lowercased.

```
str = "THIS IS STRING EXAMPLE....WOW!!!";

print str.lower();
```

Output:

this is string example....wow!!!

13. lstrip()

Returns a copy of the string in which all chars have been stripped from the beginning of the string (default whitespace characters).

```
str = " this is string example....wow!!! ";

print str.lstrip();

str = "88888888this is string example....wow!!!8888888";

print str.lstrip('8');
```

Output:

this is string example....wow!!!

this is string example....wow!!!8888888

14. max()- Returns the max alphabetical character from the string *str*.

str = "this is really a string example....wow!!!";

print "Max character: " + max(str);

str = "this is a string example....wow!!!";

print "Max character: " + max(str);

Output

Max character: y

Max character: x

15. min()-Returns the min alphabetical character from the string *str*.

str = "this-is-real-string-example....wow!!!";

print "Min character: " + min(str);

str = "this-is-a-string-example....wow!!!";

print "Min character: " + min(str);

Output:

Min character: !

Min character: !

16. replace ()-Returns a copy of the string in which the occurrences of *old have been replaced with new, optionally restricting the number of replacements to max.*

str = "this is string example....wow!!! this is really string";

print str.replace("is", "was");

print str.replace("is", "was", 3);

Output:

thwas was string example....wow!!! thwas was really string

thwas was string example....wow!!! thwas is really string

17. rstrip ()-Returns a copy of the string in which all *chars have been stripped from the end of the string (default whitespace characters)*

str = " this is string example....wow!!! ";

print str.rstrip();

str = "88888888this is string example....wow!!!8888888";

print str.rstrip('8');

Output: this is string example....wow!!!

88888888this is string example....wow!!!

18. split()-

split(str="", num=string.count(str))

Returns a list of all the words in the string, using *str as the separator (splits on all whitespace if left unspecified), optionally limiting the number of splits to num.*

str = "Line1-abcdef \nLine2-abc \nLine4-abcd";

print str.split( );

print str.split(' ', 1);

OUTPUT: ['Line1-abcdef', 'Line2-abc', 'Line4-abcd']

['Line1-abcdef', '\nLine2-abc \nLine4-abcd']

19. **swapcase()**

str = "this is string example....wow!!!";

print str.swapcase();

str = "THIS IS STRING EXAMPLE....WOW!!!";

print str.swapcase();

20. **title()**

str = "this is string example....wow!!!";

print str.title();

Output: This Is String Example....Wow!!!

21. upper() –returns a copy of the string in which all case-based characters have been uppercased.

str = "this is string example....wow!!!";

print "str.capitalize() : ", str.upper()

THIS IS STRING EXAMPLE....WOW!!!