LECTURE NOTES

ON

PYTHON PROGRAMMING
IV B. Tech I semester (CS721PE)

Prepared by

Dr.D.B.K Kamesh
Professor

M. Sreenivasulu Reddy
Assistant Professor

Mr. C. Rajeev
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MALLAREDDY ENGINEERING COLLEGE FOR WOMEN
(Autonomous Institution-UGC, Govt. of India)
Accredited by NBA & NAAC with 'A' Grade, UGC, Govt. of India
NIRF Indian Ranking, Accepted by MHRD, Govt. of India
Rank Band (6^{th}-25^{th}) by ARIIA, Accepted by MHRD, Govt. of India
Approved by AICTE, Affiliated to JNTUH, ISO 9001:2015 Certified Institution
Platinum Rated by AICTE-CII Survey, AAAA+ Rated by Digital Learning Magazine,
AAA+ Rated by Careers 360, National Ranking-Top 100 Rank band by Outlook Magazine,
3^{rd} Rank by CSR, National Ranking-Top 100 Rank band by Times News Magazine,
141 Rank by India Today-Best Engineering Colleges of India Rankings-2020.
Maisammaguda, Dhulapally, Secunderabad, Kompally-500100.

2020 – 2021

Course Name: PYTHON PROGRAMMING (Professional Elective –II)

Course Code: CS721PE

COURSE OBJECTIVES

This course will enable students to

- Learn Syntax and Semantics and create Functions in Python.
- Handle Strings and Files in Python.
- Understand Lists, Dictionaries and Regular expressions in Python.
- Implement Object Oriented Programming concepts in Python.
- Build Web Services and introduction to Network and Database Programming in Python.

COURSE OUTCOMES

The students should be able to

CO 1: Examine Python syntax and semantics and be fluent in the use of Python flow control and functions.

CO 2: Demonstrate proficiency in handling Strings and File Systems.

CO 3: Create, run and manipulate Python Programs using core data structures like Lists, Dictionaries and use Regular Expressions.

CO 4: Interpret the concepts of Object-Oriented Programming as used in Python.

CO 5: Implement exemplary applications related to Network Programming, Web Services and Databases in Python.

PROGRAM OUTCOMES

PO 1 :	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2:	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineeringsciences.
PO 3:	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4:	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5:	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO 6:	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7:	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8:	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9:	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO 10 :	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11 :	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO 12 :	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES:

CS751PC	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	√	√			√							
CO 2	√	√			√							
CO 3	√			√	√							
CO 4	√	√			√							
CO 5	√	√	√		√							

JNTUH SYLLABUS

PYTHON PROGRAMMING(Professional Elective –II)

B. Tech IV Year ISem

UNIT - I

Python Basics, Objects- Python Objects, Standard Types, Other Built-in Types, Internal Types, Standard Type Operators, Standard Type Built-in Functions, Categorizing the Standard Types, Unsupported Types.

Numbers - Introduction to Numbers, Integers, Floating Point Real Numbers, Complex Numbers, Operators, Built-in Functions, Related Modules

Sequences - Strings, Lists, and Tuples, Mapping and Set Types

UNIT - II

FILES: File Objects, File Built-in Function [open()], File Built-in Methods, File Built-in Attributes, Standard Files, Command-line Arguments, File System, File Execution, Persistent Storage Modules, Related Modules.

Exceptions: Exceptions in Python, Detecting and Handling Exceptions, Context Management, *Exceptions as Strings, Raising Exceptions, Assertions, Standard Exceptions, *Creating Exceptions, Why Exceptions (Now)?, Why Exceptions at All?, Exceptions and the sys Module, Related Modules.

Modules: Modules and Files, Namespaces, Importing Modules, Importing Module Attributes, Module Built-in Functions, Packages, Other Features of Modules

UNIT - III

Regular Expressions: Introduction, Special Symbols and Characters, Res and Python

Multithreaded Programming: Introduction, Threads and Processes, Python, Threads, and the Global Interpreter Lock, Thread Module, Threading Module, Related Modules

UNIT - IV

GUI Programming: Introduction, Tkinter and Python Programming, Brief Tour of Other GUIs, Related Modules and Other GUIs

WEB Programming: Introduction, Wed Surfing with Python, Creating Simple Web Clients, Advanced Web Clients, CGI-Helping Servers Process Client Data, Building CGI Application Advanced CGI, Web (HTTP) Servers

UNIT – V

Database Programming: Introduction, Python Database Application Programmer's Interface (DB-API), Object Relational Managers (ORMs), Related Modules

TEXT BOOK:

Core Python Programming, Wesley J. Chun, Second Edition, Pearson

INDEX

UNIT I - INTRODUCTION TO PYTHON

1.1 Python Basics

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.

1. **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
2. **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
3. **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
4. **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.
- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Features:

1. **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.
2. **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
3. **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
4. A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
5. **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

6. **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
7. **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
8. **Databases** − Python provides interfaces to all major commercial databases.
9. **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
10. **Scalable** − Python provides a better structure and support for large programs than shell scripting.

1.2 Python Objects

Python uses the object model abstraction for data storage. Any construct that contains any type of value is an object. Although Python is classified as an "object-oriented programming (OOP) language," OOP is not required to create perfectly working Python applications. You can certainly write a useful Python script without the use of classes and instances. However, Python's object syntax and architecture encourage or "provoke" this type of behavior. Let us now take a closer look at what a Python object is.

All Python objects have the following three characteristics: an identity, a type, and a value.
IDENTITY--Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the id() built-in function (BIF). This value is as close as you will get to a "memory address" in Python (probably much to the relief of some of you). Even better is that you rarely, if ever, access this value, much less care what it is at all.
TYPE--An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. You can use the type() BIF to reveal the type of a Python object. Since types are also objects in Python (did we mention that Python was object-oriented?), type() actually returns an object to you rather than a simple literal.
VALUE--Data item that is represented by an object.

All three are assigned on object creation and are read-only with one exception, the value. (For new-style types and classes, it may possible to change the type of an object, but this is not recommended for the beginner.) If an object supports updates, its value can be changed; otherwise, it is also read-only.

Whether an object's value can be changed is known as an object's mutability. These characteristics exist as long as the object does and are reclaimed when an object is deallocated.

Python supports a set of basic (built-in) data types, as well as some auxiliary types that may come into play if your application requires them. Most applications generally use the standard types and create and instantiate classes for all specialized data storage.

Object Attributes

Certain Python objects have attributes, data values or executable code such as methods, associated with them. Attributes are accessed in the dotted attribute notation, which includes the name of the associated object, and were introduced in the Core Note in Section

The most familiar attributes are

functions and methods, but some Python types have data attributes associated with them. Objects with data attributes include (but are not limited to): classes, class instances, modules, complex numbers, and files.

1.3 Standard Types

- Numbers (separate subtypes; three are integer types)
  - Integer
    - Boolean
    - Long integer
  - Floating point real number
  - Complex number
- String
- List
- Tuple
- Dictionary

We will also refer to standard types as "primitive data types" in this text because these types represent the primitive data types that Python provides

1.4 Other Built-in Types

- Type
- Null object (None)
- File
- Set/Frozenset
- Function/Method
- Module
- Class

These are some of the other types you will interact with as you develop as a Python programmer. We will also cover all of these in other chapters of this book with the exception of the type and None types, which we will discuss here.

Type Objects and the "type" Type Object

It may seem unusual to regard types themselves as objects since we are attempting to just describe all of Python's types to you in this chapter. However, if you keep in mind that an object's set of inherent behaviors and must be defined somewhere, an object's type is a logical place for this information. The amount of information necessary to describe a type cannot fit into a single string; therefore types cannot simply be strings, nor should this information be stored with the data, so we are back to types as objects.

We will formally introduce the type() BIF below, but for now, we want to let you know that you can find out the type of an object by calling type() with that object:

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

Let us look at this example more carefully. It does not look tricky by any means, but examine the return value of the call. We get the seemingly innocent output of <type 'int'>, but what you need to realize is that this is not just a simple string telling you that 42 is an integer. What you see as <type 'int'> is actually a type object. It just so happens that the string representation chosen by its implementors has a string inside it to let you know that it is an int type object.

Now you may ask yourself, so then what is the type of any type object? Well, let us find out:

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

Yes, the type of all type objects is type. The type type object is also the mother of all types and is the default metaclass for all standard Python classes. It is perfectly fine if you do not understand this now. This will make sense as we learn more about classes and types.

With the unification of types and classes in Python 2.2, type objects are playing a more significant role in both object-oriented programming as well as day-to-day object usage. Classes are now types, and instances are now objects of their respective types.

“None”, Python's Null Object

Python has a special type known as the Null object or NoneType. It has only one value, None. The type of None is NoneType. It does not have any operators or BIFs. If you are familiar with C, the closest analogy to the None type is void, while the None value is similar to the C value of NULL. (Other similar objects and values include Perl's undef and Java's void type and null value.) None has no (useful) attributes and always evaluates to having a Boolean False value.

Note: Boolean values

All standard type objects can be tested for truth value and compared to objects of the same type. Objects have inherent TRue or False values. Objects take a False value when they are empty, any numeric representation of zero, or the Null object None.

The following are defined as having false values in Python:

- None
- False (Boolean)
- Any numeric zero:
- 0 (integer)
- (float)
- 0L (long integer)
- 0.0+0.0j (complex)
- "" (empty string)
- [] (empty list)
- () (empty tuple)
- {} (empty dictionary)

Any value for an object other than those above is considered to have a true value, i.e., non-empty, non-zero, etc. User-created class instances have a false value when their nonzero (nonzero ()) or length (len ()) special methods, if defined, return a zero value.

1.5 Internal Types

- Code
- Frame
- Traceback
- Slice
- Ellipsis
- Xrange

The general application programmer would typically not interact with these objects directly, but we include them here for completeness. Please refer to the source code or Python internal and online documentation for more information.

In case you were wondering about exceptions, they are now implemented as classes. In older versions of Python, exceptions were implemented as strings.

Code Objects

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the compile() BIF. Such objects are appropriate for execution by either exec or by the eval () BIF. All this will be discussed in greater detail in Chapter 14.

Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which do contain some execution context. (The actual byte-compiled code as a code object is one attribute belonging to a function.) Along with the code object, a function's attributes also consist of the administrative support that a function requires, including its name, documentation string, default arguments, and global namespace.

Frame Objects

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment. Some of its attributes include a link to the previous stack frame, the code object (see above) that is being executed, dictionaries for the local and global namespaces, and the current instruction. Each function call results in a new frame object, and for each frame object, a C stack frame is created as well. One place where you can access a frame object is in a traceback object.

Traceback Objects

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

Traceback (innermost last):
File "<stdin>", line N?, in ???
ErrorName: error reason

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

Slice Objects

Slice objects are created using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include stride indexing, multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is sequence [start1 : end1, start2 : end2], or using the ellipsis, sequence [..., start1 : end1]. Slice objects can also be generated by the slice() BIF.

Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of sequence[starting_index : ending_index : stride].

Support for the stride element of the extended slice syntax have been in Python for a long time, but until 2.3 was only available via the C API or Jython (and previously JPython).

Here is an example of stride indexing:

```
>>> foostr = 'abcde'
>>> foostr[::-1] 'edcba'
>>> foostr[::-2] 'eca'
>>> foolist = [123, 'xba', 342.23, 'abc']
>>> foolist[::-1]
['abc', 342.23, 'xba', 123]
```

Ellipsis Objects

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (...). Like the Null object None, ellipsis objects also have a single name, Ellipsis, and have a Boolean TRue value at all times.

XRange Objects

XRange objects are created by the BIF xrange(), a sibling of the range() BIF, and used when memory is limited and when range() generates an unusually large data set.

For an interesting side adventure into Python types, we invite the reader to take a look at the typesmodule in the standard Python library.

1.6 Standard Type Operators

Object Value Comparison

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types. Comparisons yield Boolean TRue or False values, based on the validity of the comparison expression. (If you are using Python prior to 2.3 when the Boolean type was introduced, you will see integer values 1 for TRue and 0 for False.)

Standard Type Value Comparison Operators

Operator	Function
expr1 < expr2	expr1 is less than expr2
expr1 > expr2	expr1 is greater than expr2
expr1 <= expr2	expr1 is less than or equal to expr2
expr1 >= expr2	expr1 is greater than or equal to expr2
expr1 == expr2	expr1 is equal to expr2
expr1 != expr2	expr1 is not equal to expr2 (C-style)
expr1 <> expr2	expr1 is not equal to expr2 (ABC/Pascal-style)[a]

This "not equal" sign will be phased out in future version of Python. Use != instead.

Note that comparisons performed are those that are appropriate for each data type. In other words, numeric types will be compared according to numeric value in sign and magnitude, strings will compare lexicographically, etc.

```
>>> 2 == 2
True
>>> 2.46 <= 8.33
True
>>> 5+4j >= 2-3j
True
>>> 'abc' == 'xyz' False
```

```
>>> 'abc' > 'xyz' False
>>> 'abc' < 'xyz' True
>>> [3, 'abc'] == ['abc', 3]
False
>>> [3, 'abc'] == [3, 'abc']
True
```

Also, unlike many other languages, multiple comparisons can be made on the same line, evaluated in left-to-right order:

```
>>> 3 < 4 < 7  # same as ( 3 < 4 ) and ( 4 < 7 ) True
>>> 4 > 3 == 3          # same as ( 4 > 3 ) and ( 3 == 3 ) True
>>> 4 < 3 < 5 != 2 < 7
False
```

We would like to note here that comparisons are strictly between object values, meaning that the comparisons are between the data values and not the actual data objects themselves. For the latter, we will defer to the object identity comparison operators described next.

Object Identity Comparison

In addition to value comparisons, Python also supports the notion of directly comparing objects themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

In order to understand this, you will have to think of variables as linking to objects now and be less concerned with the values themselves. Let us take a look at three examples.

Example 1: foo1 and foo2 reference the same object

```
foo1 = foo2 = 4.3
```

When you look at this statement from the value point of view, it appears that you are performing a multiple assignment and assigning the numeric value of 4.3 to both the foo1 and foo2 variables. This is true to a certain degree, but upon lifting the covers, you will find that a numeric object with the contents or value of 4.3 has been created. Then that object's reference is assigned to both foo1 and foo2, resulting in both foo1 and foo2 aliased to the same object. Figure 4-1 shows an object with two references.

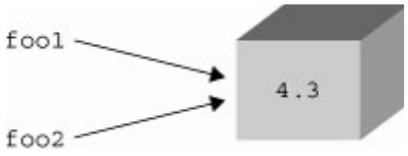

Figure 1.1 : foo1 and foo2 reference the same object

Example 2: foo1 and foo2 reference the same object
foo1 = 4.3 foo2 = foo1

This example is very much like the first: A numeric object with value 4.3 is created, then assigned to one variable. When foo2 = foo1 occurs, foo2 is directed to the same object as foo1 since Python deals with objects by passing references. foo2 then becomes a new and additional reference for the original value. So both foo1 and foo2 now point to the same object. The same figure above applies here as well.

Example 3: foo1 and foo2 reference different objects
foo1 = 4.3
foo2 = 1.3 + 3.0

This example is different. First, a numeric object is created, then assigned to foo1. Then a second numeric object is created, and this time assigned to foo2. Although both objects are storing the exact same value, there are indeed two distinct objects in the system, with foo1 pointing to the first, and foo2 being a reference to the second. Figure 4-2 shows we now have two distinct objects even though both objects have the same value.

Figure 1.2: foo1 and foo2 reference different objects

Why did we choose to use boxes in our diagrams? Well, a good way to visualize this concept is to imagine a box (with contents inside) as an object. When a variable is assigned an object, that creates a "label" to stick on the box, indicating a reference has been made. Each time a new reference to the same object is made, another sticker is put on the box. When references are abandoned, then a label is removed. A box can be "recycled" only when all the labels have been peeled off the box. How does the system keep track of how many labels are on a box?
Each object has associated with it a counter that tracks the total number of references that exist to that object.

This number simply indicates how many variables are "pointing to" any particular object Performing a check such as a is b is an equivalent expression to id(a) == id(b).

The object identity comparison operators all share the same precedence level and are presented in Table

Standard Type Object Identity Comparison Operators

Operator	Function
obj1 is obj2	obj1 is the same object as obj2
obj1 is not obj2	obj1 is not the same object as obj2

In the example below, we create a variable, and then another that points to the same object.

```
>>> a = [ 5, 'hat', -9.3]
>>> b = a
>>> a is b True
>>> a is not b False
>>>
>>> b = 2.5e-5
>>> b 2.5e-005
>>> a
[5, 'hat', -9.3]
>>> a is b False
>>> a is not b True
```

Both the is and not identifiers are Python keywords.

Note: Interning

In the above examples with the foo1 and foo2 objects, you will notice that we use floating point values rather than integers. The reason for this is although integers and strings are immutable objects, Python sometimes caches them to be more efficient. This would have caused the examples to appear that Python is not creating a new object when it should have.

For example:

```
>>> a = 1
>>> id(a) 8402824
>>> b = 1
>>> id(b) 8402824
>>>
>>> c = 1.0
>>> id(c) 8651220
>>> d = 1.0
>>> id(d) 8651204
```

In the above example, a and b reference the same integer object, but c and d do not reference the same float object. If we were purists, we would want a and b to work just like c and d because we really did ask to create a new integer object rather than an alias, as in b = a.

Python caches or interns only simple integers that it believes will be used frequently in any Python application. At the time of this writing, Python interns integers in the range(-1, 100) but this is subject to change, so do not code your application to expect this.

In Python 2.3, the decision was made to no longer intern strings that do not have at least one reference outside of the "interned strings table." This means that without that reference, interned strings are nolonger immortal and subject to garbage collection like everything else.

A BIF introduced in 1.5 to request interning of strings, intern(), has now been deprecated as a result.

Boolean

Expressions may be linked together or negated using the Boolean logical operators and, or, and not, all of which are Python keywords. These Boolean operations are in highest-to-lowest order of precedence in Table 4.3. The not operator has the highest precedence and is immediately one level below all the comparison operators. The and and or operators follow, respectively.

Standard Type Boolean Operators

Operator	Function
not expr	Logical NOT of expr (negation)
expr1 and expr2	Logical AND of expr1 and expr2 (conjunction)
expr1 orexpr2	Logical OR of expr1 and expr2 (disjunction)

```
>>> x, y = 3.1415926536, -1024
>>> x < 5.0
True
>>> not (x < 5.0) False
>>> (x < 5.0) or (y > 2.718281828)
True
>>> (x < 5.0) and (y > 2.718281828)
False
>>> not (x is y) True
```

Earlier, we introduced the notion that Python supports multiple comparisons within one expression. These expressions have an implicit and operator joining them together.

```
>>> 3 < 4 < 7  # same as "( 3 < 4 ) and ( 4 < 7 )" True.
```

1.7 Standard Type Build-in Functions

Along with generic operators, which we have just seen, Python also provides some BIFs that can be applied to all the basic object types: cmp(),repr(),str(),type(), and the single reverse or back quotes (``) operator, which is functionally equivalent to repr().

Standard Type Built-in Functions

Function	Operation
cmp(obj1, obj2)	Compares obj1 and obj2, returns integer i where: i < 0 if obj1 < obj2 i > 0 if obj1 > obj2 i == 0 if obj1 == obj2
repr(obj) or `obj`	Returns evaluatable string representation of obj
str(obj)	Returns printable string representation of obj
type(obj)	Determines type of obj and return type object

type()

We now formally introduce type(). In Python versions earlier than 2.2, type() is a BIF. Since that release, it has become a "factory function." We will discuss these later on in this chapter, but for now, you may continue to think of type() as a BIF. The syntax for type() is:

type(object):type() takes an object and returns its type. The return value is a type object.

```
>>> type(4)     # int type
<type 'int'>
>>>
>>> type('Hello World!')       # string type
<type 'string'>
>>>
>>> type(type(4))       # type type
<type 'type'>
```

In the examples above, we take an integer and a string and obtain their types using the type() BIF; in order to also verify that types themselves are types, we call type() on the output of a type() call.

Note the interesting output from the type() function. It does not look like a typical Python data type, i. e., a number or string, but is something enclosed by greater-than and less-than signs. This syntax is generally a clue that what you are looking at is an object. Objects may implement a printable string representation; however, this is not always the case. In these scenarios where there is no easy way to "display" an object, Python "pretty-prints" a string representation of the object. The format is usually of the form: <object_something_or_another>. Any object displayed in this manner generally gives the object type, an object ID or location, or other pertinent information.

cmp():

The cmp() BIF CoMPares two objects, say, obj1 and obj2, and returns a negative number (integer) if obj1 is less than obj2, a positive number if obj1 is greater than obj2, and zero if obj1 is equal to obj2. Notice the similarity in return values as C's strcmp(). The comparison used is the one that applies for that type of object, whether it be a standard type or a user-created class; if the latter, cmp() will call the class's special cmp () method. Here are some samples of using the cmp() BIF with numbers and strings.

```
>>> a, b = -4, 12
>>> cmp(a,b)
-1
>>> cmp(b,a) 1
>>> b = -4
>>> cmp(a,b) 0
>>>
>>> a, b = 'abc', 'xyz'
>>> cmp(a,b)
-23
>>> cmp(b,a) 23
>>> b = 'abc'
>>> cmp(a,b) 0
```

We will look at using cmp() with other objects later.

str() and repr() (and `` Operator)

The str() STRing and repr() REPResentation BIFs or the single back or reverse quote operator (``) come in very handy if the need arises to either re-create an object through evaluation or obtain a human-readable view of the contents of objects, data values, object types, etc. To use these operations, a Python object is provided as an argument and some type of string representation of that object is returned. In the examples that follow, we take some random Python types and convert them to their string representations.

```
>>> str(4.53-2j) '(4.53-2j)'
>>> str(1) '1'
>>> str(2e10) '20000000000.0'
>>> str([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>> repr([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>> `[0, 5, 9, 9]`
'[0, 5, 9, 9]'
```

Although all three are similar in nature and functionality, only repr() and `` do exactly the same thing, and using them will deliver the "official" string representation of an object that can be evaluated as a valid Python expression (using the eval() BIF). In contrast, str() has the job of delivering a "printable" string representation of an object, which may not necessarily be acceptable by eval(), but will look nice in a print statement. There is a caveat that while most return values from repr() can be evaluated, not all can:

```
>>> eval(`type(type))`)
File "<stdin>", line 1 eval(`type(type))`)
SyntaxError: invalid syntax
```

The executive summary is that repr( ) is Python-friendly while str() produces human-friendly output. However, with that said, because both types of string representations coincide so often, on many occasions all three return the exact same string.

Note: Why have both repr() and ``?

Occasionally in Python, you will find both an operator and a function that do exactly the same thing. One reason why both an operator and a function exist is that there are times where a function may be more useful than the operator, for example, when you are passing around executable objects like functions and where different functions may be called depending on the data item. Another example is the double-star (**) and pow() BIF, which performs "x to the y power" exponentiation for x ** y or pow(x,y).

type() and isinstance()

Python does not support method or function overloading, so you are responsible for any "introspection" of the objects that your functions are called with.

What's in a name? Quite a lot, if it is the name of a type. It is often advantageous and/or necessary to base pending computation on the type of object that is received. Fortunately, Python provides a BIF just for that very purpose. type() returns the type for any Python object, not just the standard types. Usingthe interactive interpreter, let us take a look at some examples of what type() returns when we give it various objects.

```
>>> type('')
<type 'str'>
>>>
>>> s = 'xyz'
>>> type(s)
<type 'str'>
>>>
>>> type(100)
<type 'int'>
>>> type(0+0j)
<type 'complex'>
>>> type(0L)
```

```
<type 'long'>
>>> type(0.0)
<type 'float'>
>>>
>>> type([])
<type 'list'>
>>> type(())
<type 'tuple'>
>>> type({})
<type 'dict'>
>>> type(type)
<type 'type'>
>>>
>>> class Foo: pass    # new-style class
...
>>> foo = Foo()
>>> class Bar(object): pass    # new-style class
...
>>> bar = Bar()
>>>
>>> type(Foo)
<type 'classobj'>
>>> type(foo)
<type 'instance'>
>>> type(Bar)
<type 'type'>
>>> type(bar)
<class ' main .Bar'>
```

Types and classes were unified in Python 2.2. You will see output different from that above if you are using a version of Python older than 2.2:

```
>>> type('')
<type 'string'>
>>> type(0L)
<type 'long int'>
>>> type({})
<type 'dictionary'>
>>> type(type)
<type 'builtin_function_or_method'>
>>> type(Foo)# assumes Foo created as in above
<type 'class'>
>>> type(foo) # assumes foo instantiated also
<type 'instance'>
```

In addition to type(), there is another useful BIF called isinstance().

Example

We present a script in Example that shows how we can use isinstance() and type() in a runtime environment. We follow with a discussion of the use of type() and how we migrated to using isinstance () instead for the bulk of the work in this example.

Example: Checking the Type (typechk.py)

The function displayNumType() takes a numeric argument and uses the type() built-in to indicate its type (or "not a number," if that is the case).

```
1 #!/usr/bin/env python 2
3       def displayNumType(num):
4       print num, 'is',
5       if isinstance(num, (int, long, float, complex)):
6       print 'a number of type:', type(num). name
7       else:
8       print 'not a number at all!!' 9
10 displayNumType(-69)
11 displayNumType(99999999999999999999L)
12      displayNumType(98.6)
13      displayNumType(-5.2+1.9j)
14      displayNumType('xxx')
```

Running typechk.py, we get the following output:

```
-69 is a number of type: int
99999999999999999999 is a number of type: long
98.6 is a number of type: float
(-5.2+1.9j) is a number of type: complex
xxx is not a number at all!!
```

Reducing Number of Function Calls

If we take a closer look at our code, we see a pair of calls to type(). As you know, we pay a small price each time a function is called, so if we can reduce that number, it will help with performance.

An alternative to comparing an object's type with a known object's type (as we did above and in the example below) is to utilize the types module, which we briefly mentioned earlier in the chapter. If we do that, then we can use the type object there without having to "calculate it." We can then change our code to only having one call to the type() function:

```
>>> import types
>>> if type(num) == types.IntType.
```

Object Value Comparison versus Object Identity Comparison

We discussed object value comparison versus object identity comparison earlier in this chapter, and if you realize one key fact, then it will become clear that our code is still not optimal in terms of performance. During runtime, there is always only one type object that represents an integer. In other words, type(0), type(42), type(-100) are always the same object: <type 'int'> (and this is also the same object as types.IntType).

If they are always the same object, then why do we have to compare their values since we already know they will be the same? We are "wasting time" extracting the values of both objects and comparing them if they are the same object, and it would be more optimal to just compare the objects themselves. Thus we have a migration of the code above to the following:

```
if type(num) is types.IntType... # or type(0)
```

Does that make sense? Object value comparison via the equal sign requires a comparison of their values, but we can bypass this check if the objects themselves are the same. If the objects are different,then we do not even need to check because that means the original variable must be of a different type (since there is only one object of each type). One call like this may not make a difference, but if there are many similar lines of code throughout your application, then it starts to add up.

Reduce the Number of Lookups

This is a minor improvement to the previous example and really only makes a difference if your application performs makes many type comparisons like our example. To actually get the integer type object, the interpreter has to look up the types name first, and then within that module's dictionary, find IntType. By using from-import, you can take away one lookup:

```
from types import IntType
if type(num) is IntType ...
```

Convenience and Style

The unification of types and classes in 2.2 has resulted in the expected rise in the use of the isinstance () BIF. We formally introduce isinstance() in Chapter 13 (Object-Oriented Programming), but we will give you a quick preview now.

This Boolean function takes an object and one or more type objects and returns true if the object in question is an instance of one of the type objects. Since types and classes are now the same, int is now a type (object) and a class. We can use isinstance() with the built-in types to make our if statement more convenient and readable:

```
if isinstance(num, int)...
```

Using isinstance() along with type objects is now also the accepted style of usage when introspecting objects' types, which is how we finally arrive at our updated typechk.py application above. We also get the added bonus of isinstance() accepting a tuple of type objects to check against our object with instead of having an if-elif-else if we were to use only type().

1.8 Categorizing the Standard Types

Since Python 2.2 with the unification of types and classes, all of the built-in types are now classes, and with that, all of the "conversion" built-in functions like int(), type(), list(), etc., are now factory functions.

This means that although they look and act somewhat like functions, they are actually class names, and when you call one, you are actually instantiating an instance of that type, like a factory producing a good.

The following familiar factory functions were formerly built-in functions:

- int(), long(), float(), complex()
- str(), unicode(), basestring()
- list(), tuple()
- type()

Other types that did not have factory functions now do. In addition, factory functions have been added for completely new types that support the new-style classes.

The following is a list of both types of factory functions:

- dict()
- bool()
- set(), frozenset()
- object()
- classmethod()
- staticmethod()
- super()
- property()
- file()

1.9 Unsupported Types

Before we explore each standard type, we conclude this chapter by giving a list of types that are not supported by Python.

char or byte

Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.

pointer

Since Python manages memory for you, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the id() BIF. Since you have no control over this value, it's a moot point. However, under Python's covers, everything is a pointer.

int versus short versus long

Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, e.g., C's int, short, and long. For the record, Python's integers are implemented as C longs. Also, since there is a close relationship between Python's int and long types, users have even fewer things to worry about. You only need to use a single type, the Python integer. Even when the size of an integer is exceed, for example, multiplying two very large numbers, Python automatically gives you a long back instead of overflowing with an error.

float versus double:

C has both a single precision float type and double-precision double type. Python's float type is actually a C double. Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types. For those wanting more accuracy and willing to give up a wider range of numbers, Python has a decimal floating point number too, but you have to import the decimal module to use the Decimal type. Floats are always estimations. Decimals are exact and arbitrary precision. Decimals make sense concerning things like money where the values are exact. Floats make sense for things that are estimates anyway, such as weights, lengths, and other measurements.

1.10 Introduction to Numbers

Numbers provide literal or scalar storage and direct access. A number is also an immutable type, meaning that changing or updating its value results in a newly allocated object. This activity is, of course, transparent to both the programmer and the user, so it should not change the way the application is developed.

Python has several numeric types: "plain" integers, long integers, Boolean, double-precision floating point real numbers, decimal floating point numbers, and complex numbers.

How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable:

```
anInt = 1
aLong = -9999999999999999L
aFloat = 3.14159265358979323846264338327 95
aComplex = 1.23+4.56J
```

How to Update Numbers

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether. We put quotes around update because you are not really changing the value of the original variable. Because numbers are immutable, you are just making a new number and reassigning the reference. Do not be fooled by what you were taught about how variables contain values that allow you to update them. Python's object model is more specific than that.

When we learned programming, we were taught that variables act like boxes that hold values. In Python, variables act like pointers that point to boxes. For immutable types, you do not change the contents of the box, you just point your pointer at a new box. Every time you assign another number to a variable, you are creating a new object and assigning it. (This is true for all immutable types, not just numbers.)

```
anInt += 1
aFloat = 2.718281828
```

How to Remove Numbers

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, just use the del statement (introduced in Section 3.5.6). You can no longer use the variable name, once removed, unless you assign it to a new object; otherwise, you will cause a NameError exception to occur.

```
del anInt
del aLong, aFloat, aComplex
```

Okay, now that you have a good idea of how to create and update numbers, let us take a look at Python's four numeric types.

1.11 Integers

Python has several types of integers. There is the Boolean type with two possible values. There are the regular or plain integers: generic vanilla integers recognized on most systems today. Python also has a long integer size; however, these far exceed the size provided by C longs . We will take a look at these types of integers, followed by a description of operators and built-in functions applicable only to Python integer types.

Boolean

The Boolean type was introduced in Python 2.3. Objects of this type have two possible values, BooleanTRue and False. We will explore Boolean objects toward the end of this chapter in Section 5.7.1.

Standard (Regular or Plain) Integers

Python's "plain" integers are the universal numeric type. Most machines (32-bit) running Python will provide a range of -231 to 231-1, that is -2, 147,483,648 to 2,147,483,647. If Python is compiled on a 64-bit system with a 64-bit compiler, then the integers for that system will be 64-bit. Here are some examples of Python integers:

```
0101   84      -237   0x80   017     -680    -0X92
```

Python integers are implemented as (signed) longs in C. Integers are normally represented in base 10 decimal format, but they can also be specified in base 8 or base 16 representation. Octal values have a "0" prefix, and hexadecimal values have either "0x" or "0X" prefixes.

Long Integers

The first thing we need to say about Python long integers (or longs for short) is not to get them confused with longs in C or other compiled languagesthese values are typically restricted to 32- or 64- bit sizes.

whereas Python longs are limited only by the amount of (virtual) memory in your machine. In other words, they can be very L-O-N-G longs.

Longs are a superset of integers and are useful when your application requires integers that exceed the range of plain integers, meaning less than -231 or greater than 231-1.

Use of longs is denoted by the letter "L", uppercase (L) or lowercase (l), appended to the integer's numeric value. Values can be expressed in decimal, octal, or hexadecimal. The following are examples of longs:

16384L -0x4E8L 017L -2147483648l 052144364L
299792458l 0xDECADEDEADBEEFBADFEEDDEAL -5432101234L

Core Style: Use uppercase "L" with long integers

Although Python supports a case-insensitive "L" to denote longs, we recommend that you use only the uppercase "L" to avoid confusion with the number one (1). Python will display only longs with a capital "L ." As integers and longs are slowly being unified, you will only see the "L" with evaluatable string representations (repr()) of longs.
Printable string representations (str()) will not have the "L ."

```
>>> aLong = 999999999l
>>> aLong 999999999L
>>> print aLong 999999999
```

1.12 Floating Point Real Numbers

Floats in Python are implemented as C doubles, double precision floating point real numbers, values that can be represented in straightforward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about ± 10308.25 in range), and the final bit to the sign. That all sounds fine and dandy; however, the actual degree of precision you will receive (along with the range and overflow handling) depends completely on the architecture of the machine as well as the implementation of the compiler that built your Python interpreter.

Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase (e) or uppercase (E). Positive (+) or negative (-) signs between the "e" and the exponent indicate the sign of the exponent. Absence of such a sign indicates a positive exponent. Here are some floating point values:

0.0	-777.	1.6	-5.555567119	96e3 * 1.0
4.3e25	9.384e-23	-2.172818	float(12)	1.000000001
3.1416	4.2E-10	-90.	6.022e23	-1.609E-19

1.13 Complex Numbers

A long time ago, mathematicians were absorbed by the following equation: x2 = -1
The reason for this is that any real number (positive or negative) multiplied by itself results in a positive number. How can you multiply any number with itself to get a negative number? No such real number exists. So in the eighteenth century, mathematicians invented something called an imaginary number i (or) j, depending on what math book you are reading) such that:

Basically a new branch of mathematics was created around this special number (or concept), and now imaginary numbers are used in numerical and mathematical applications. Combining a real number with an imaginary number forms a single entity known as a complex number. A complex number is any ordered pair of floating point real numbers (x, y) denoted by x + yj where x is the real part and y is the imaginary part of a complex number.

It turns out that complex numbers are used a lot in everyday math, engineering, electronics, etc. Because it became clear that many researchers were reinventing this wheel quite often, complex numbers became a real Python data type long ago in version 1.4.

Here are some facts about Python's support of complex numbers:

- Imaginary numbers by themselves are not supported in Python (they are paired with a real part of 0.0 to make a complex number)
- Complex numbers are made up of real and imaginary parts
- Syntax for a complex number: real+imagj
- Both real and imaginary components are floating point values
- Imaginary part is suffixed with letter "J" lowercase (j) or uppercase (J)

The following are examples of complex numbers:

64.375+1j	4.23-8.5j	0.23-8.55j	1.23e-045+6.7e+089j
6.23+1.5j	-1.23-875J 0+1j	9.80665-8.31441J	-.0224+0j

1.14 Operators

Numeric types support a wide variety of operators, ranging from the standard type of operators to operators created specifically for numbers, and even some that apply to integer types only.

Mixed-Mode Operations

It may be hard to remember, but when you added a pair of numbers in the past, what was important was that you got your numbers correct. Addition using the plus (+) sign was always the same. In programming languages, this may not be as straightforward because there are different types of numbers.

When you add a pair of integers, the + represents integer addition, and when you add a pair of floating point numbers, the + represents double-precision floating point addition, and so on. Our little description extends even to non-numeric types in Python. For example, the + operator for strings represents concatenation, not addition, but it uses the same operator! The point is that for each data type that supports the + operator, there are different pieces of functionality to "make it all work," embodying the concept of overloading.

Now, we cannot add a number and a string, but Python does support mixed mode operations strictly between numeric types. When adding an integer and a float, a choice has to be made as to whether integer or floating point addition is used. There is no hybrid

operation. Python solves this problem using something called numeric coercion. This is the process whereby one of the operands is converted to the same type as the other before the operation. Python performs this coercion by following some basic rules.

To begin with, if both numbers are the same type, no conversion is necessary. When both types are different, a search takes place to see whether one number can be converted to the other's type. If so, the operation occurs and both numbers are returned, one having been converted. There are rules that must be followed since certain conversions are impossible, such as turning a float into an integer, or converting a complex number to any non-complex number type.

Coercions that are possible, however, include turning an integer into a float (just add ".0") or converting any non-complex type to a complex number (just add a zero imaginary component, e.g., "0j"). The rules of coercion follow from these two examples: integers move toward float, and all move toward complex. The Python Language Reference Guide describes the coerce() operation in the following manner.

- If either argument is a complex number, the other is converted to complex;
- Otherwise, if either argument is a floating point number, the other is converted to floating point;
- Otherwise, if either argument is a long, the other is converted to long;
- Otherwise, both must be plain integers and no conversion is necessary (in the upcoming diagram, this describes the rightmost arrow).

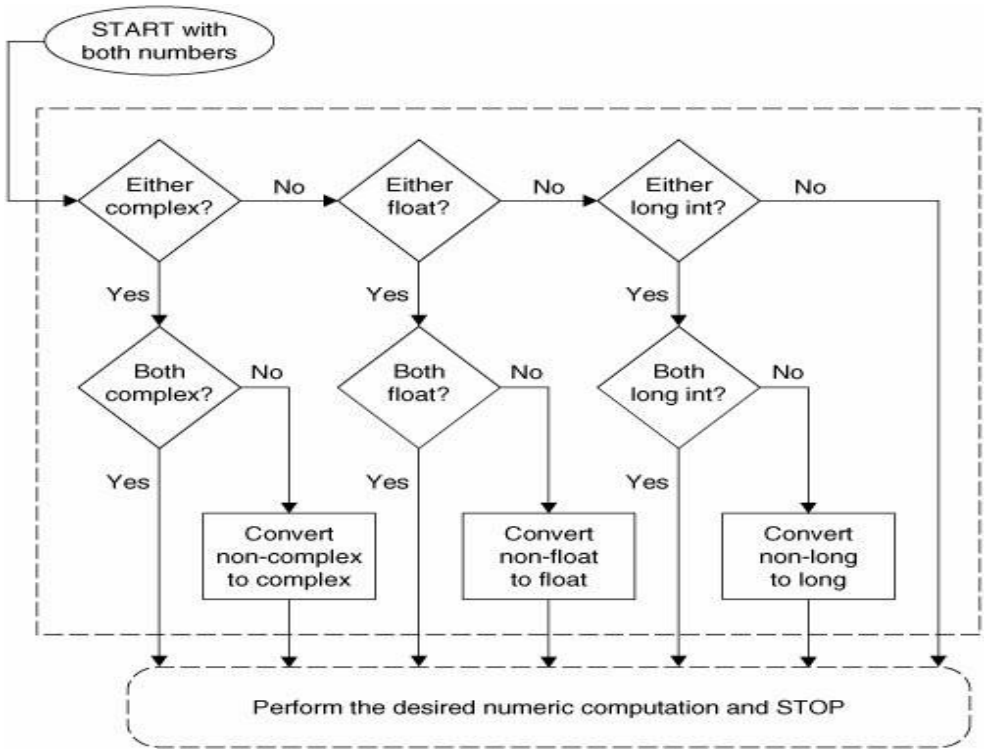

Figure 1.3 : Numeric Coercion

Automatic numeric coercion makes life easier for the programmer because he or she does not have to worry about adding coercion code to his or her application. If explicit coercion is desired, Python does provide the coerce() built-in.

The following is an example showing you Python's automatic coercion. In order to add the numbers (one integer, one float), both need to be converted to the same type. Since float is the superset, the integer is coerced to a float before the operation happens, leaving the result as a float:

```
>>> 1 + 4.5
5.5
```

- Standard Type Operators
- Numeric Type (Arithmetic) Operators
- Bit Operators (Integer-Only)

1.15 Built-in Functions

Built-in and Factory Functions are

- Standard Type Functions
- Numeric Type Functions
- Integer-Only Functions

1.16 Related Modules

There are a number of modules in the Python standard library that add on to the functionality of the operators and built-in functions for numeric types.

1.17 Sequences Introduction

Sequence types all share the same access model: ordered set with sequentially indexed offsets to get to each element. Multiple elements may be selected by using the slice operators, which we will explore in this chapter. The numbering scheme used starts from zero (0) and ends with one less than the length of the sequence the reason for this is because we began at 0. Fig illustrates how sequence items are stored.

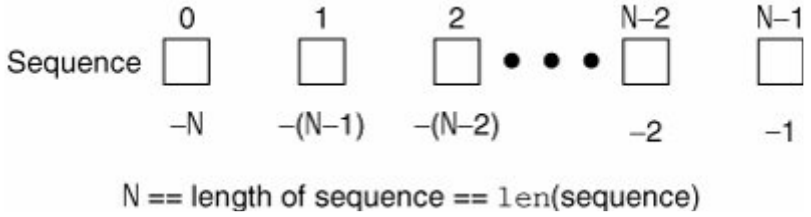

Fig 1.4: sequence representation

1.18 Strings

Strings are among the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. This contrasts with most other shell-type scripting languages, which use single quotes for literal strings and double quotes to allow escaping of characters. Python uses the "raw string" operator to create literal quotes, so no differentiation is necessary. Other languages such as C use single quotes for characters and double quotes for strings.

Python does not have a character type; this is probably another reason why single and double quotes are treated the same.

Nearly every Python application uses strings in one form or another. Strings are a literal or scalar type, meaning they are treated by the interpreter as a singular value and are not containers that hold other Python objects. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

With the unification of types and classes in 2.2, there are now actually three types of strings in Python. Both regular string (str) and Unicode string (unicode) types are actually subclassed from an abstract class called basestring. This class cannot be instantiated, and if you try to use the factory function to make one, you get this:

```
>>> basestring('foo')
Traceback (most recent call last): File "<stdin>", line 1, in <module>
TypeError: The basestring type cannot be instantiated
```

How to Create and Assign Strings

Creating strings is as simple as using a scalar value or having the str() factory function make one and assigning it to a variable:

```
>>> aString = 'Hello World!' # using single quotes
>>> anotherString = "Python is cool!" # double quotes
>>> print aString        # print, no quotes! Hello World!
>>> anotherString       # no print, quotes! 'Python is cool!'
>>> s = str(range(4))  # turn list to string
>>> s
'[0, 1, 2, 3]'
```

How to Access Values (Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0] 'H'
>>> aString[1:5] 'ello'
>>> aString[6:] 'World!'
```

How to Update 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 altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString 'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

Like numbers, strings are not mutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string.

However, as you can see above, there is nothing wrong with piecing together parts of your old string into a new string.

How to Remove Characters and Strings

To repeat what we just said, strings are immutable, so you cannot remove individual characters from an existing string. What you can do, however, is to empty the string, or to put together another string that drops the pieces you were not interested in.

Let us say you want to remove one letter from "Hello World!"...the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString 'Helo World!'
```

To clear or remove a string, you assign an empty string or use the del statement, respectively:

```
>>> aString = ''
>>> aString ''
>>> del aString
```

In most applications, strings do not need to be explicitly deleted. Rather, the code defining the stringeventually terminates, and the string is eventually deallocated.

1.19 Lists and Tuples

Lists

Like strings, lists provide sequential storage through an index offset and access to single or consecutive elements through slices. However, the comparisons usually end there. Strings consist only of characters and are immutable (cannot change individual elements), while lists are flexible container objects that hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy, too, as we see in the following examples.

The objects that you can place in a list can include standard types and objects as well as user-defined ones. Lists can contain different types of objects and are more flexible than an array of C structs or Python arrays (available through the external array module) because arrays are restricted to containing objects of a single type.

Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will.

Tuples share many of the same characteristics of lists and although we have a separate section on tuples, many of the examples and list functions are applicable to tuples as well.

The key difference is that tuples are immutable, i.e., read-only, so any operators or functions that allow updating lists, such as using the slice operator on the left-hand side of an assignment, will not be valid for tuples.

How to Create and Assign Lists

Creating lists is as simple as assigning a value to a variable. You handcraft a list (empty or with elements) and perform the assignment. Lists are delimited by surrounding square brackets ([ ]). You can also use the factory function.

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> print anotherList
[None, 'something to see here']
>>> aListThatStartedEmpty = []
>>> print aListThatStartedEmpty []
>>> list('foo') ['f', 'o', 'o']
```

How to Access Values in Lists

Slicing works similar to strings; use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> aList[0] 123
>>> aList[1:4]
['abc', 4.56, ['inner', 'list']]
>>> aList[:3] [123, 'abc', 4.56]
>>> aList[3][1]
'list'
```

How to Update Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method:

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2] 4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>
>>> anotherList.append("hi, i'm new here")
>>> print anotherList
[None, 'something to see here', "hi, i'm new here"]
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty ['not empty anymore']
```

How to Remove List Elements and Lists

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

You can also use the pop() method to remove and return a specific object from a list. Normally, removing an entire list is not something application programmers do. Rather, they tend to let it go out of scope (i.e., program termination, function call completion, etc.) and be deallocated, but if they do want to explicitly remove an entire list, they use the del statement:

```
del aList
```

Tuples

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally, there is a more significant difference, and that is the fact that tuples are immutable. Because of this, tuples can do something that lists cannot do .be a dictionary key. Tuples are also the default when dealing with a group of objects.

Our usual modus operandi is to present the operators and built-in functions for the more general objects, followed by those for sequences and conclude with those applicable only for tuples, but because tuples share so many characteristics with lists, we would be duplicating much of our description from the previous section. Rather than providing much repeated information, we will differentiate tuples from lists as they apply to each set of operators and functionality, then discuss immutability and other features unique to tuples.

How to Create and Assign Tuples

Creating and assigning tuples are practically identical to creating and assigning lists, with the exception of empty tuplesthese require a trailing comma (,) enclosed in the tuple delimiting parentheses (( )) to prevent them from being confused with the natural grouping operation of parentheses.

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
>>> print anotherTuple
(None, 'something to see here')
```

```
>>> emptiestPossibleTuple = (None,)
>>> print emptiestPossibleTuple (None,)
>>> tuple('bar') ('b', 'a', 'r')
```

How to Access Values in Tuples

Slicing works similarly to lists. Use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> aTuple[1:4]
('abc', 4.56, ['inner', 'tuple'])

>>> aTuple[:3] (123, 'abc', 4.56)
>>> aTuple[3][1] 'tuple'
```

How to Update Tuples

Like numbers and strings, tuples are immutable, which means you cannot update them or change values of tuple elements. In Sections 6.2 and 6.3.2, we were able to take portions of an existing string to create a new string. The same applies for tuples.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the del statement to reduce an object's reference count. It will be deallocated when that count is zero. Keep in mind that most of the time one will just let an object go out of scope rather than using del, a rare occurrence in everyday Python programming.

```
del aTuple
```

1.20 Mapping and Set Types

Mapping Type: Dictionaries

Dictionaries are the sole mapping type in Python. Mapping objects have a one-to-many correspondence between hashable values (keys) and the objects they represent (values). They are similar to Perl hashes and can be generally considered as mutable hash tables. A dictionary object itself is mutable and is yet another container type that can store any number of Python objects, including other container types.

What makes dictionaries different from sequence type containers like lists and tuples is the way the data are stored and accessed.

Sequence types use numeric keys only (numbered sequentially as indexed offsets from the beginning of the sequence). Mapping types may use most other object types as keys; strings are the most common. Unlike sequence type keys, mapping keys are often, if not directly, associated with the data value that is stored. But because we are no longer using "sequentially ordered" keys with mapping types, we are left with an unordered collection of data. As it turns out, this does not hinder our use because mapping types do not require a numeric value to index into a container to obtain the desired item. With a key, you are "mapped" directly to your value, hence the term "mapping type." The reason why they are commonly referred to as hash tables is because that is the exact type of object that dictionaries are one of Python's most powerful data types.

How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict1, dict2
({}, {'port': 80, 'name': 'earth'})
```

In Python versions 2.2 and newer, dictionaries may also be created using the factory function dict(). We discuss more examples later when we take a closer look at dict(), but here's a sneak peek for now:

```
>>> fdict = dict((['x', 1], ['y', 2]))
>>> fdict
{'y': 2, 'x': 1}
```

In Python versions 2.3 and newer, dictionaries may also be created using a very convenient built-in method for creating a "default" dictionary whose elements all have the same value (defaulting to None if not given), fromkeys():

```
>>> ddict = {}.fromkeys(('x', 'y'), -1)
>>> ddict
{'y': -1, 'x': -1}
>>> edict = {}.fromkeys(('foo', 'bar'))
>>> edict #output:{'foo': None, 'bar': None}
```

How to Access Values in Dictionaries

To traverse a dictionary (normally by key), you only need to cycle through its keys, like this:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2.keys():
...         print 'key=%s, value=%s' % (key, dict2[key])
        key=name, value=earth key=port, value=80
```

Beginning with Python 2.2, you no longer need to use the keys() method to extract a list of keys to loop over. Iterators were created to simplify accessing of sequence-like objects such as dictionaries and files. Using just the dictionary name itself will cause an iterator over that dictionary to be used in a for loop:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2:
...     print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth key=port, value=80
```

To access individual dictionary elements, you use the familiar square brackets along with the key to obtain its value:

```
>>> dict2['name'] 'earth'
>>>
>>> print 'host %s is running on port %d' % \
...     (dict2['name'], dict2['port']) host earth is running on port 80
```

Dictionary dict1 defined above is empty while dict2 has two data items. The keys in dict2 are 'name' and 'port', and their associated value items are 'earth' and 80, respectively. Access to the value is through the key, as you can see from the explicit access to the 'name' key.

If we attempt to access a data item with a key that is not part of the dictionary, we get an error:

```
>>> dict2['server'] Traceback (innermost last):
File "<stdin>", line 1, in ?
KeyError: server
```

In this example, we tried to access a value with the key 'server' which, as you know from the code above, does not exist. The best way to check if a dictionary has a specific key is to use the dictionary's has_key() method, or better yet, the in or not in operators starting with version 2.2. The has_key() method will be obsoleted in future versions of Python, so it is best to just use in or not in.

We will introduce all of a dictionary's methods below. The Boolean has_key() and the in and not in operators are Boolean, returning true if a dictionary has that key and False otherwise. (In Python versions preceding Boolean constants [older than 2.3], the values returned are 1 and 0, respectively.)

```
>>> 'server' in dict2 # or dict2.has_key('server') False
>>> 'name' in dict # or dict2.has_key('name') True
>>> dict2['name'] 'earth'
```

Here is another dictionary example mixing the use of numbers and strings as keys:

```
>>> dict3 = {}
>>> dict3[1] = 'abc'
>>> dict3['1'] = 3.14159
>>> dict3[3.2] = 'xyz'
>>> dict3
{3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

Rather than adding each key-value pair individually, we could have also entered all the data for dict3 at the same time:

```
dict3 = {3.2: 'xyz', 1: 'abc', '1': 3.14159}
```

Creating the dictionary with a set key-value pair can be accomplished if all the data items are known in advance (obviously). The goal of the examples using dict3 is to illustrate the variety of keys that you can use. If we were to pose the question of whether a key for a particular value should be allowed to change, you would probably say, "No." Right?

Not allowing keys to change during execution makes sense if you think of it this way: Let us say that you created a dictionary element with a key and value. Somehow during execution of your program, the key changed, perhaps due to an altered variable. When you went to retrieve that data value again with the original key, you got a KeyError (since the key changed), and you had no idea how to obtain your value now because the key had somehow been altered. For this reason, keys must be hashable, so numbers and strings are fine, but lists and other dictionaries are not. (See Section 7.5.2 for why keys must be hashable.)

How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a key-value pair), modifying an existing entry, or deleting an existing entry (see below for more details on removing an entry).

```
>>> dict2['name'] = 'venus' # update existing entry
>>> dict2['port'] = 6969       # update existing entry
>>> dict2['arch'] = 'sunos5' # add new entry
>>> print 'host %(name)s is running on port %(port)d' % dict2
host venus is running on port 6969
```

If the key does exist, then its previous value will be overridden by its new value. The print statement above illustrates an alternative way of using the string format operator (%), specific to dictionaries. Using the dictionary argument, you can shorten the print request somewhat because naming of the dictionary occurs only once, as opposed to occurring for each element using a tuple argument.You may also add the contents of an entire dictionary to another dictionary by using the update() built- in method.

How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary. However, if you really want to "remove" an entire dictionary, use the del statement. Here are some deletion examples for dictionaries and dictionary elements:

del dict2['name'] # remove entry with key 'name' dict2.clear() # remove all entries in dict1

del dict2 # delete entire dictionary dict2.pop('name') # remove & return entry w/key

Set Types

In mathematics, a set is any collection of distinct items, and its members are often referred to as set elements. Python captures this essence in its set type objects. A set object is an unordered collection of hashable values. Yes, set members would make great dictionary keys. Mathematical sets translate to Python set objects quite effectively and testing for set membership and operations such as union and intersection work in Python as expected.

Like other container types, sets support membership testing via in and not in operators, cardinality using the len() BIF, and iteration over the set membership using for loops. However, since sets are unordered, you do not index into or slice them, and there are no keys used to access a value.

There are two different types of sets available, mutable (set) and immutable (frozenset). As you can imagine, you are allowed to add and remove elements from the mutable form but not the immutable. Note that mutable sets are not hashable and thus cannot be used as either a dictionary key or as an element of another set. The reverse is true for frozen sets, i.e., they have a hash value and can be used as a dictionary key or a member of a set.

How to Create and Assign Set Types

There is no special syntax for sets like there is for lists ([ ]) and dictionaries ({ }) for example. Lists and dictionaries can also be created with their corresponding factory functions list() and dict(), and that is also the only way sets can be created, using their factory functions set() and frozenset():

```
>>> s = set('cheeseshop')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's'])
>>> t = frozenset('bookshop')
>>> t
frozenset(['b', 'h', 'k', 'o', 'p', 's'])
>>> type(s)
<type 'set'>
>>> type(t)
```

<type 'frozenset'>
>>> len(s) 6
>>> len(s) == len(t) True
>>> s == t
False

How to Access Values in Sets

You are either going to iterate through set members or check if an item is a member (or not) of a set:

>>> 'k' in s False
>>> 'k' in t True
>>> 'c' not in t True
>>> for i in s:
... print i
...
c e h o p s

How to Update Sets

You can add and remove members to and from a set using various built-in methods and operators:

>>> s.add('z')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's', 'z'])
>>> s.update('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y', 'z'])
>>> s.remove('z')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
>>> s -= set('pypi')
>>> s
set(['c', 'e', 'h', 'o', 's'])

As mentioned before, only mutable sets can be updated. Any attempt at such operations on immutable sets is met with an exception:

>>> t.add('z')
Traceback (most recent call last): File "<stdin>", line 1, in ?
AttributeError: 'frozenset' object has no attribute 'add'

How to Remove Set Members and Sets

We saw how to remove set members above. As far as removing sets themselves, like any Python object, you can let them go out of scope or explicitly remove them from the current namespace with del. If the reference count goes to zero, then it is tagged for garbage collection.

>>> del s

UNIT II – FILES & EXCEPTIONS IN PYTHON

2.1 Files - File Objects

File objects can be used to access not only normal disk files, but also any other type of "file" that uses that abstraction. Once the proper "hooks" are installed, you can access other objects with file-style interfaces in the same manner you would access normal files.

You will find many cases where you are dealing with "file-like" objects as you continue to develop your Python experience. Some examples include "opening a URL" for reading a Web page in real-time and launching a command in a separate process and communicating to and from it like a pair of simultaneously open files, one for write and the other for read.

The open() built-in function (see below) returns a file object that is then used for all succeeding operations on the file in question. There are a large number of other functions that return a file or file- like object. One primary reason for this abstraction is that many input/output data structures prefer to adhere to a common interface. It provides consistency in behavior as well as implementation. Operating systems like Unix even feature files as an underlying and architectural interface for communication.
Remember, files are simply a contiguous sequence of bytes. Anywhere data need to be sent usually involves a byte stream of some sort, whether the stream occurs as individual bytes or blocks of data

2.2 File Built-in Function [open()]

As the key to opening file doors, the open() [and file()] built-in function provides a general interface to initiate the file input/output (I/O) process. The open() BIF returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or raises an IOError exceptionwe will cover errors and exceptions in the next chapter.

The basic syntax of the open() built-in function is:
file_object = open(file_name, access_mode='r', buffering=-1)

The file_name is a string containing the name of the file to open. It can be a relative or absolute/full pathname. The access_mode optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes 'r,' 'w,'or 'a,' representing read, write, and append, respectively. A 'U' mode also exists for universal NEWLINE support (see below).

Any file opened with mode 'r' or 'U' must exist. Any file opened with 'w' will be truncated first if it exists, and then the file is (re)created. Any file opened with 'a' will be opened for append. All writes to files opened with 'a' will be from end-of-file, even if you seek elsewhere during access. If the file does not exist, it will be created, making it the same as if you opened the file in 'w' mode. If you are a C programmer, these are the same file open modes used for the C library function fopen().

There are other modes supported by fopen() that will work with Python's open(). These include the '+' for read-write access and 'b' for binary access.

One note regarding the binary flag: 'b' is antiquated on all Unix systems that are POSIX-compliant (including Linux) because they treat all files as binary files, including text files. Here is an entry from the Linux manual page for fopen(), from which the Python open () function is derived:

The mode string can also include the letter "b" either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI C3.159-1989 ("ANSI C") and has no effect; the "b" is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the "b" may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

2.3 File Built-in Methods

Once open() has completed successfully and returned a file object, all subsequent access to the file transpires with that "handle." File methods come in four different categories: input, output, movement within a file, which we will call "intra-file motion," and miscellaneous. A summary of all file methods can be found in Table 9.3. We will now discuss each category.

Input

The read() method is used to read bytes directly into a string, reading at most the number of bytes indicated. If no size is given (the default value is set to integer -1) or size is negative, the file will be read to the end. It will be phased out and eventually removed in a future version of Python.

The readline() method reads one line of the open file (reads all bytes until a line-terminating character like NEWLINE is encountered). The line, including termination character(s), is returned as a string. Like read(), there is also an optional size option, which, if not provided, defaults to -1, meaning read until the line-ending characters (or EOF) are found. If present, it is possible that an incomplete line is returned if it exceeds size bytes.

The readlines() method does not return a string like the other two input methods. Instead, it reads all (remaining) lines and returns them as a list of strings. Its optional argument, sizhint, is a hint on the maximum size desired in bytes. If provided and greater than zero, approximately sizhint bytes in whole lines are read (perhaps slightly more to round up to the next buffer size) and returned as a list.

In Python 2.1, a new type of object was used to efficiently iterate over a set of lines from a file: the xreadlines object (found in the xreadlines module). Calling file.xreadlines() was equivalent to xreadlines. xreadlines(file). Instead of reading all the lines in at once, xreadlines() reads in chunks at a time, and thus were optimal for use with for loops in a memory-conscious way. However, with the introduction of iterators and the new file iteration in Python 2.3, it was no longer necessary to have an xreadlines() method because it is the same as using iter(file), or in a for loop, is replaced by for eachLine in file. Easy come, easy go.

Another odd bird is the readinto() method, which reads the given number of bytes into a writable buffer object, the same type of object returned by the unsupported buffer() built-in function. (Since buffer() is not supported, neither is readinto().)

Output

The write() built-in method has the opposite functionality as read() and readline(). It takes a string that can consist of one or more lines of text data or a block of bytes and writes the data to the file.

The writelines() method operates on a list just like readlines(), but takes a list of strings and writes them out to a file. Line termination characters are not inserted between each line, so if desired, they must be added to the end of each line before writelines() is called.

Note that there is no "writeline()" method since it would be equivalent to calling write() with a single line string terminated with a NEWLINE character.

2.4 File Built-in Attributes

File objects also have data attributes in addition to methods. These attributes hold auxiliary data related to the file object they belong to, such as the file name (file.name), the mode with which the file was opened (file.mode), whether the file is closed (file.closed), and a flag indicating whether an additional space character needs to be displayed before successive data items when using the print statement (file.softspace). Table 9.4 lists these attributes along with a brief description of each.

Attributes for File Objects

File Object Attribute	Description
file.closed	TRue if file is closed and False otherwise
file.encoding	Encoding that this file useswhen Unicode strings are written to file, they will be converted to byte strings using file.encoding; a value of None indicates that the system default encoding for converting Unicode strings should be used
file.mode	Access mode with which file was opened
file.name	Name of file
file.newlines	None if no line separators have been read, a string consisting of one type of line separator, or a tuple containing all types of line termination characters read so far
file.softspace	0 if space explicitly required with print, 1 otherwise; rarely used by the programmergenerally for internal use only.

2.5 Standard Files

There are generally three standard files that are made available to you when your program starts. These are standard input (usually the keyboard), standard output (buffered output to the monitor or display), and standard error (unbuffered output to the screen). (The "buffered" or "unbuffered" output refers to that third argument to open()). These files are named stdin, stdout, and stderr and take their names from the C language. When we say these files are "available to you when your program starts," that means that these files are pre-opened for you, and access to these files may commence once you have their file handles.

Python makes these file handles available to you from the sys module. Once you import sys, you have access to these files as sys.stdin, sys.stdout, and sys.stderr. The print statement normally outputs to sys.stdout while the raw_input() built-in function receives its input from sys.stdin.

Just remember that since sys.* are files, you have to manage the line separation characters. The print
statement has the built-in feature of automatically adding one to the end of a string to output.

2.6 Command-line Arguments

The sys module also provides access to any command-line arguments via sys.argv. Command-line arguments are those arguments given to the program in addition to the script name on invocation. Historically, of course, these arguments are so named because they are given on the command line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case. Most IDEs provide a separate window with which to enter your "command-line arguments." These, in turn, will be passed into the program as if you started your application from the command line.

Those of you familiar with C programming may ask, "Where is argc?" The names "argc" and "argv" stand for "argument count" and "argument vector," respectively. The argv variable contains an array of strings consisting of each argument from the command line while the argc variable contains the number of arguments entered. In Python, the value for argc is simply the number of items in the sys.argv list, and the first element of the list, sys.argv[0], is always the program name.

Summary:

- sys.argv is the list of command-line arguments
- len(sys.argv) is the number of command-line arguments(aka argc) Let us create a small test program called argv.py with the following lines: import sys

```
print 'you entered', len(sys.argv), 'arguments...'
print 'they were:', str(sys.argv)
```

Here is an example invocation and output of this script:

```
$ argv.py 76 tales 85 hawk you entered 5 arguments...
they were: ['argv.py', '76', 'tales', '85', 'hawk']
```

Are command-line arguments useful? Commands on Unix-based systems are typically programs that take input, perform some function, and send output as a stream of data. These data are usually sent as input directly to the next program, which does some other type of function or calculation and sends the new output to another program, and so on. Rather than saving the output of each program and potentially taking up a good amount of disk space, the output is usually "piped" into the next program as its input.

This is accomplished by providing data on the command line or through standard input. When a program displays or sends output to the standard output file, the result would be displayed on the screenunless that program is also "piped" to another program, in which case that standard output file is really the standard input file of the next program. I assume you get the drift by now!

Command-line arguments allow a programmer or administrator to start a program perhaps with different behavioral characteristics. Much of the time, this execution takes place in the middle of the night and runs as a batch job without human interaction. Command-line arguments and program options enable this type of functionality.

As long as there are computers sitting idle at night and plenty of work to be done, there will always be a need to run programs in the background on our veryexpensive "calculators."

Python has two modules to help process command-line arguments. The first (and original), getopt is easier but less sophisticated, while optparse, introduced in Python 2.3, is more powerful library and is much more object-oriented than its predecessor. If you are just getting started, we recommend getopt, but once you outgrow its feature set, then check out optparse.

2.7 File System

Access to your file system occurs mostly through the Python os module. This module serves as the primary interface to your operating system facilities and services from Python. The os module is actually a front-end to the real module that is loaded, a module that is clearly operating systemdependent. This "real" module may be one of the following: posix (Unix-based, i.e., Linux, MacOS X, *BSD, Solaris, etc.), nt (Win32), mac (old MacOS), dos (DOS), os2 (OS/2), etc. You should never import those modules directly. Just import os and the appropriate module will be loaded, keeping all the underlying work hidden from sight. Depending on what your system supports, you may not have access to some of the attributes, which may be available in other operating system modules.

In addition to managing processes and the process execution environment, the os module performs most of the major file system operations that the application developer may wish to take advantage of. These features include removing and renaming files, traversing the directory tree, and managing file accessibility.

2.8 File Execution

Whether we want to simply run an operating system command, invoke a binary executable, or another type of script (perhaps a shell script, Perl, or Tcl/Tk), this involves executing another file somewhere else on the system. Even running other Python code may call for starting up another Python interpreter, although that may not always be the case. Please proceed there if you are interested in how to start other programs, perhaps even communicating with them, and for general information regarding Python's execution environment.

2.9 Persistent Storage Modules

In many of the exercises in this text, user input is required. After many iterations, it may be somewhat frustrating being required to enter the same data repeatedly. The same may occur if you are entering a significant amount of data for use in the future. This is where it becomes useful to have persistent storage, or a way to archive your data so that you may access them at a later time instead of having to re-enter all of that information. When simple disk files are no longer acceptable and full relational database management systems (RDBMSs) are overkill, simple persistent storage fills the gap. The majority of the persistent storage modules deals with storing strings of data, but there are ways to archive Python objects as well.

pickle and marshal Modules

Python provides a variety of modules that implement minimal persistent storage. One set of modules (marshal and pickle) allows for pickling of Python objects.

Pickling is the process whereby objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms. Pickling is also known as flattening, serializing, or marshalling. Another set of modules (dbhash/bsddb, dbm, gdbm, dumbdbm) and their "manager" (anydbm) can provide persistent storage of Python strings only.

As we mentioned before, both marshal and pickle can flatten Python objects. These modules do not provide "persistent storage" per se, since they do not provide a namespace for the objects, nor can they provide concurrent write access to persistent objects. What they can do, however, is to pickle Python objects to allow them to be stored or transmitted. Storage, of course, is sequential in nature (you store or transmit objects one after another). The difference between marshal and pickle is that marshal can handle only simple Python objects (numbers, sequences, mapping, and code) while pickle can transform recursive objects, objects that are multi-referenced from different places, and user-defined classes and instances. The pickle module is also available in a turbo version called cPickle, which implements all functionality in C.

DBM-style Modules

The *db* series of modules writes data in the traditional DBM format. There are a large number of different implementations: dbhash/bsddb, dbm, gdbm, and dumbdbm. If you are particular about any specific DBM module, feel free to use your favorite, but if you are not sure or do not care, the generic anydbm module detects which DBM-compatible modules are installed on your system and uses the "best" one at its disposal. The dumbdbm module is the most limited one, and is the default used if none of the other packages is available. These modules do provide a namespace for your objects, using objects that behave similar to a combination of a dictionary object and a file object. The one limitation of these systems is that they can store only strings. In other words, they do not serialize Python objects.

shelve Module

Finally, we have a somewhat more complete solution, the shelve module. The shelve module uses the anydbm module to find a suitable DBM module, then uses cPickle to perform the pickling process. The shelve module permits concurrent read access to the database file, but not shared read/write access. This is about as close to persistent storage as you will find in the Python standard library. There may beother external extension modules that implement "true" persistent storage. The diagram in Figure shows the relationship between the pickling modules and the persistent storage modules, and how the shelve object appears to be the best of both worlds.

2.10 Related Modules

The fileinput module iterates over a set of input files and reads their contents one line at a time, allowing you to iterate over each line, much like the way the Perl (<>) operator works without any provided arguments. File names that are not explicitly given will be assumed to be provided from the command-line.

The glob and fnmatch modules allow for file name pattern-matching in the good old-fashioned Unix shell- style, for example, using the asterisk (*) wildcard character for all string matches and the (?) for matching single characters.

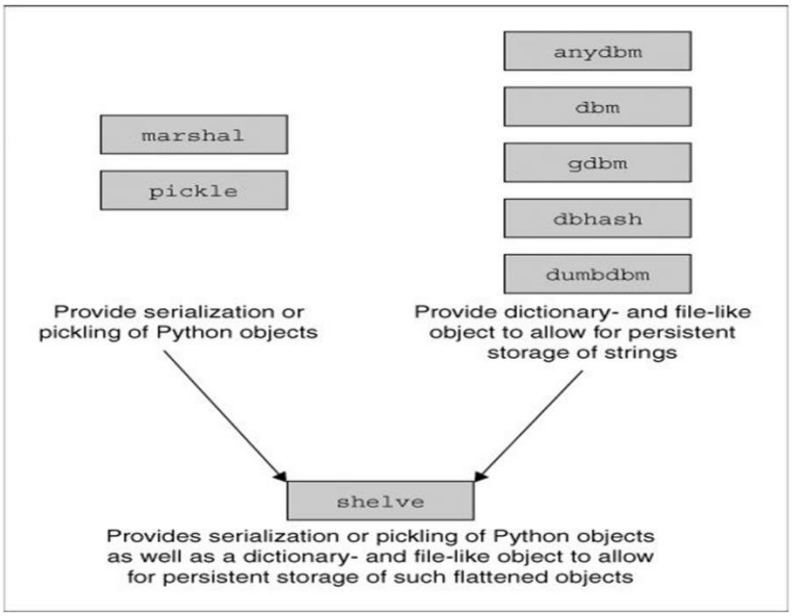

Figure: 2.1 Python modules for serialization and persistency

2.11 Exceptions - Exceptions in Python

As we were going through some of the, you no doubt noticed what happens when your program "crashes" or terminates due to unresolved errors. A "trace back" notice appears along with a notice containing as much diagnostic information as the interpreter can give you, including the error name, reason, and perhaps even the line number near or exactly where the error occurred. All errors have a similar format, regardless of whether running within the Python interpreter or standard script execution, providing a consistent error interface. All errors, whether they be syntactical or logical, result from behavior incompatible with the Python interpreter and cause exceptions to be raised.

Let us take a look at some exceptions now.
NameError: attempt to access an undeclared variable

```
>>> foo
Traceback (innermost last):
File "<stdin>", line 1, in ? NameError: name 'foo' is not defined
```

NameError indicates access to an uninitialized variable. The offending identifier was not found in the Python interpreter's symbol table. We will be discussing namespaces in the next two chapters, but as an introduction, regard them as "address books" linking names to objects. Any object that is accessible should be listed in a namespace. Accessing a variable entails a search by the interpreter, and if the name requested is not found in any of the namespaces, a NameError exception will be generated.

ZeroDivisionError: division by any numeric zero

```
>>> 1/0
Traceback (innermost last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Our example above used floats, but in general, any numeric division-by-zero will result in a ZeroDivisionError exception.

SyntaxError: Python interpreter syntax error
>>> for
File "<string>", line 1
for
SyntaxError: invalid syntax

SyntaxError exceptions are the only ones that do not occur at run-time. They indicate an improperly constructed piece of Python code which cannot execute until corrected. These errors are generated at compile-time, when the interpreter loads and attempts to convert your script to Python bytecode. These may also occur as a result of importing a faulty module.

IndexError: request for an out-of-range index for sequence
>>> aList = []
>>> aList[0]
Traceback (innermost last): File "<stdin>", line 1, in ?
IndexError: list index out of range

IndexError is raised when attempting to access an index that is outside the valid range of a sequence.

KeyError: request for a non-existent dictionary key
>>> aDict = {'host': 'earth', 'port': 80}
>>> print aDict['server'] Traceback (innermost last):
File "<stdin>", line 1, in ?
KeyError: server

Mapping types such as dictionaries depend on keys to access data values. Such values are not retrieved if an incorrect/nonexistent key is requested. In this case, a KeyErroris raised to indicate such an incident has occurred.

IOError: input/output error
>>> f = open("blah") Traceback (innermost last):
File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'blah'

Attempting to open a nonexistent disk file is one example of an operating system input/output (I/O) error. Any type of I/O error raises an IOError exception.

AttributeError: attempt to access an unknown object attribute
>>> class myClass(object):
... pass
>>> myInst = myClass()
>>> myInst.bar = 'spam'
>>> myInst.bar 'spam'
>>> myInst.foo
Traceback (innermost last): File "<stdin>", line 1, in ?
AttributeError: foo

In our example, we stored a value in myInst.bar, the bar attribute of instance myInst. Once an attribute has been defined, we can access it using the familiar dotted-attribute notation, but if it has not, as in our case with the foo (non-)attribute, an AttributeError occurs.

2.12 Detecting and Handling Exceptions

Exceptions can be detected by incorporating them as part of a try statement. Any code suite of a Try statement will be monitored for exceptions.

There are two main forms of the Try statement: Try-except and try-finally. These statements are mutually exclusive, meaning that you pick only one of them. A try statement can be accompanied by one or more except clauses, exactly one finally clause, or a hybrid try-except-finally combination.

try-except statements allow one to detect and handle exceptions. There is even an optional else clause for situations where code needs to run only when no exceptions are detected. Meanwhile, TRy-finally statements allow only for detection and processing of any obligatory cleanup (whether or not exceptions occur), but otherwise have no facility in dealing with exceptions. The combination, as you might imagine, does both.

try-except Statement

The TRy-except statement (and more complicated versions of this statement) allows you to define a section of code to monitor for exceptions and also provides the mechanism to execute handlers for exceptions.

The syntax for the most general try-except statement is given below. It consists of the keywords along with the try and except blocks (try_suite and except_suite) as well as optionally saving the reason of failure:

```
try:
try_suite        # watch for exceptions here
except Exception[, reason]:
except_suite    # exception-handling code
```

Let us give one example, then explain how things work. We will use our IOError example from above. We can make our code more robust by adding a try-except "wrapper" around the code:

```
>>> try:
...        f = open('blah', 'r')
... except IOError, e:
...        print 'could not open file:', e
...
```

could not open file: [Errno 2] No such file or directory

As you can see, our code now runs seemingly without errors. In actuality, the same IOError still occurred when we attempted to open the nonexistent file. The difference? We added code to both detect and handle the error. When the IOError exception was raised, all we told the interpreter to do was to output a diagnostic message. The program continues and does not "bomb out" as our earlier examplea minor illustration of the power of exception handling. So what is really happening codewise?

During runtime, the interpreter attempts to execute all the code within the try statement. If an exception does not occur when the code block has completed, execution resumes past the except statement. When the specified exception named on the except statement does occur, we save the reason, and control flow immediately continues in the handler (all remaining code in the TRy clause is skipped) where we display our error message along with the cause of the error.

2.13 Context Management

The unification of TRy-except and TRy-finally as described above makes programs more "Pythonic," meaning, among many other characteristics, simpler to write and easier to read. Python already does a great job at hiding things under the covers so all you have to do is worry about how to solve the problem you have. (Can you imagine porting a complex Python application into C++ or Java?!?)

Another example of hiding lower layers of abstraction is the with statement, made official as of Python 2.6. (It was introduced in 2.5 as a preview and to serve warnings for those applications using with as an identifier that it will become a keyword in 2.6. To use this feature in 2.5, you must import it with from future importwith_statement.)

Like try-except-finally, the with statement, has a purpose of simplifying code that features the common idiom of using the TRy-except and try-finally pairs in tandem. The specific use that the with statement targets is when TRy-except and try-finally are used together in order to achieve the sole allocation of a shared resource for execution, then releasing it once the job is done. Examples include files (data, logs, database, etc.), threading resources and synchronization primitives, database connections, etc.

However, instead of just shortening the code and making it easier to use like try-except-finally, the with statement's goal is to remove the TRy, except, and finally keywords and the allocation and release code from the picture altogether. The basic syntax of the with statement looks like this:

```
with context_expr [as var]:
with_suite
```

It looks quite simple, but making it work requires some work under the covers. The reason is it not as simple as it looks is because you cannot use the with statement merely with any expression in Python. It only works with objects that support what is called the context management protocol. This simply means that only objects that are built with "context management" can be used with a with statement. We will describe what that means soon.

Now, like any new video game hardware, when this feature was released, some folks out there took the time to develop new games for it so that you can play when you open the box. Similarly, there were already some Python objects that support the protocol. Here is a short list of the first set:

- file
- decimal.Context
- tHRead.LockType
- threading.Lock
- threading.RLock
- threading.Condition
- tHReading.Semaphore

Since files are first on the list and the simplest example, here is a code snippet of what it looks like to use a with statement:

```
with open('/etc/passwd', 'r') as f:
for eachLine in f:
# ...do stuff with eachLine or f...
```

2.14 Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

For backward compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option -Xwill provide you with the standard exceptions as strings. This feature will be obsolete beginning with Python 1.6.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions within the scope of this book and have removed it. (You may find the original text in prior editions of this book.) The only point of relevance and the final thought is a caution: You may use an external or third- party module, which may still have string exceptions. String exceptions are a bad idea anyway. One reader vividly recalls seeing Linux RPM exceptions with spelling errors in the exception text.

2.15 Raising Exceptions

The interpreter was responsible for raising all of the exceptions we have seen so far. These exist as a result of encountering an error during execution. A programmer writing an API may also wish to throw an exception on erroneous input, for example, so Python provides a mechanism for the programmer to explicitly generate an exception: the raise statement.

raise Statement Syntax and Common Usage

The raise statement is quite flexible with the arguments it supports, translating to a large number of different formats supported syntactically. The general syntax for raise is:

```
raise [SomeException [, args [, traceback]]]
```

The first argument, SomeException, is the name of the exception to raise. If present, it must either be a string, class, or instance (more below). SomeException must be given if any of the other arguments (args or traceback) are present.

The second expression contains optional args (aka parameters, values) for the exception. This value is either a single object or a tuple of objects. When exceptions are detected, the exception arguments are always returned as a tuple. If args is a tuple, then that tuple represents the same set of exception arguments that are given to the handler. If args is a single object, then the tuple will consist solely of this one object (i.e., a tuple with one element). In most cases, the single argument consists of a string indicating the cause of the error. When a tuple is given, it usually equates to an error string, an error number, and perhaps an error location, such as a file, etc.

The final argument, TRaceback, is also optional (and rarely used in practice), and, if present, is the traceback object used for the exceptionnormally a traceback object is newly created when an exception is raised. This third argument is useful if you want to reraise an exception (perhaps to point to the previous location from the current). Arguments that are absent are represented by the value None.

The most common syntax used is when SomeException is a class. No additional parameters are ever required, but in this case, if they are given, they can be a single object argument, a tuple of arguments, or an exception class instance. If the argument is an instance, then it can be an instance of the given class or a derived class (subclassed from a pre-existing exception class). No additional arguments (i.e., exception arguments) are permitted if the argument is an instance.

2.16 Assertions

Assertions are diagnostic predicates that must evaluate to Boolean true; otherwise, an exception is raised to indicate that the expression is false. These work similarly to the assert macros, which are part of the C language preprocessor, but in Python these are runtime constructs (as opposed to precompile directives).

If you are new to the concept of assertions, no problem. The easiest way to think of an assertion is to liken it to a raise-if statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, introduced back in version 1.5.

assert Statement

The assert statement evaluates a Python expression, taking no action if the assertion succeeds (similar to a pass statement), but otherwise raising an AssertionError exception. The syntax for assert is:

assert expression [, arguments]

Here are some examples of the use of the assert statement:

```
assert 1 == 1
assert 2 + 2 == 2 * 2
assert len(['my list', 12]) < 10
assert range(3) == [0, 1, 2]
```

AssertionError exceptions can be caught and handled like any other exception using the TRy-except statement, but if not handled, they will terminate the program and produce a traceback similar to the following:

```
>>> assert 1 == 0
Traceback (innermost last): File "<stdin>", line 1, in ?
AssertionError
```

As with the raise statement we investigated in the previous section, we can provide an exception argument to our assert command:

```
>>> assert 1 == 0, 'One does not equal zero silly!' Traceback (innermost last):
File "<stdin>", line 1, in ?
AssertionError: One does not equal zero silly!
```

Here is how we would use a TRy-except statement to catch an AssertionErrorexception:

```
try:
assert 1 == 0, 'One does not equal zero silly!'
except AssertionError, args:
print '%s: %s' % (args. class . name , args)
```

Executing the above code from the command line would result in the following output:

AssertionError: One does not equal zero silly!

To give you a better idea of how assert works, imagine how the assert statement may be implemented in Python if written as a function. It would probably look something like this:

```
def assert(expr, args=None):
if debug and not expr:
raise AssertionError, args
```

The first if statement confirms the appropriate syntax for the assert, meaning that expr should be an expression. We compare the type of expr to a real expression to verify. The second part of the function evaluates the expression and raises AssertionError, if necessary. The built-in variable debug is 1 under normal circumstances, 0 when optimization is requested (command-line option -O).

2.17 Standard Exceptions

As of Python 2.5, all exceptions are new-style classes and are ultimately subclassed from BaseException. At this release, SystemExit and KeyboardInterrupt were taken out of the hierarchy for Exception and moved up to being under BaseException. This is to allow statements like except Exception to catch all errors and not program exit conditions.

From Python 1.5 through Python 2.4.x, exceptions were classic classes, and prior to that, they were strings. String-based exceptions are no longer acceptable constructs and are officially deprecated beginning with 2.5, where you will not be able to raise string exceptions. In 2.6, you cannot catch them.

There is also a requirement that all new exceptions be ultimately subclassed from BaseException so that all exceptions will have a common interface. This will transition will begin with Python 2.7 and continue through the remainder of the 2.x releases.

2.18 Creating Exceptions

Although the set of standard exceptions is fairly wide-ranging, it may be advantageous to create your own exceptions. One situation is where you would like additional information from what a standard or module-specific exception provides. We will present two examples, both related to IOError.

IOError is a generic exception used for input/output problems, which may arise from invalid file access or other forms of communication. Suppose we wanted to be more specific in terms of identifying the source of the problem. For example, for file errors, we want to have a FileError exception that behaves like IOError, but with a name that has more meaning when performing file operations.

Another exception we will look at is related to network programming with sockets. The exception generated by the socket module is called socket.error and is not a built-in exception. It is subclassed from the generic Exception exception. However, the exception arguments from socket.error closely resemble those of IOError exceptions, so we are going to define a new exception called NetworkError, which subclasses from IOError but contains at least the information provided by socket.error.

2.19 Why Exceptions (Now)?

There is no doubt that errors will be around as long as software is around. The difference in today's fast- paced computing world is that our execution environments have changed, and so has our need to adapt error-handling to accurately reflect the operating context of the software that we develop. Modern-day applications generally run as self-contained graphical user interfaces (GUIs) or in a client/server architecture such as the Web.

The ability to handle errors at the application level has become even more important recently in that users are no longer the only ones directly running applications. As the Internet and online electronic commerce become more pervasive, Web servers will be the primary users of application software. This means that applications cannot just fail or crash outright anymore, because if they do, system errors translate to browser errors, and these in turn lead to frustrated users. Losing eyeballs means losing advertising revenue and potentially significant amounts of irrecoverable business.

If errors do occur, they are generally attributed to some invalid user input. The execution environment must be robust enough to handle the application-level error and be able to produce a user-level error message. This must translate to a "non-error" as far as the Web server is concerned because the application must complete successfully, even if all it does is return an error message to present to the user as a valid Hypertext Markup Language (HTML) Web page displaying the error.

If you are not familiar with what I am talking about, does a plain Web browser screen with the big black words saying, "Internal Server Error" sound familiar? How about a fatal error that brings up a pop-up that declares "Document contains no data"? As a user, do either of these phrases mean anything to you? No, of course not (unless you are an Internet software engineer), and to the average user, they are an endless source of confusion and frustration. These errors are a result of a failure in the execution of an application. The application either returns invalid Hypertext Transfer Protocol (HTTP) data or terminates fatally, resulting in the Web server throwing its hands up into the air, saying, "I give up!"

This type of faulty execution should not be allowed, if at all possible. As systems become more complex and involve more apprentice users, additional care should be taken to ensure a smooth user application experience. Even in the face of an error situation, an application should terminate successfully, as to not affect its execution environment in a catastrophic way. Python's exception handling promotes mature and correct programming.

2.20 Why Exceptions at All?

If the above section was not motivation enough, imagine what Python programming might be like without program level exception handling. The first thing that comes to mind is the loss of control client programmers have over their code. For example, if you created an interactive application that allocates and utilizes a large number of resources, if a user hit ^C or other keyboard interrupt, the application would not have the opportunity toper form cleanup.

2.21 Exceptions and the sys Module

An alternative way of obtaining exception information is by accessing the exc_info() function in the sys module. This function provides a 3-tuple of information, more than what we can achieve by simply using only the exception argument. Let us see what we get using sys.exc_info():

```
>>> try:
...     float('abc123')
... except:
...     import sys
...     exc_tuple = sys.exc_info()
...
>>> print exc_tuple
(<class exceptions.ValueError at f9838>, <exceptions. ValueError instance at
122fa8>,
<traceback object at 10de18>)
>>>

>>> for eachItem in exc_tuple:
...     print eachItem
...
exceptions.ValueError
invalid literal for float(): abc123
<traceback object at 10de18>
```

What we get from sys.exc_info() in a tuple are:

- exc_type: exception class object
- exc_value: (this) exception class instance object
- exc_traceback: traceback object

The first two items we are familiar with: the actual exception class and this particular exception's instance (which is the same as the exception argument which we discussed in the previous section). The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred.

In older versions of Python, these three values were available in the sys module as sys.exc_type, sys. exc_value, and sys.exc_traceback. Unfortunately, these three are global variables and not thread-safe. We recommend using sys.exc_info() instead. All three will be phased out and eventually removed in a future version of Python.

2.22 Related Modules

Exception-Related Standard Library Modules

Module	Description
exceptions	Built-in exceptions (never need to import this module)
contextlib	Context object utilities for use with the with statement
sys	Contains various exception-related objects and functions (see sys.ex*)

2.23 Modules - Modules and Files

A module allows you to logically organize your Python code. When code gets to be large enough, the tendency is to break it up into organized pieces that can still interact with one another at a functioning level. These pieces generally have attributes that have some relation to one another, perhaps a single class with its member data variables and methods, or maybe a group of related, yet independently operating functions. These pieces should be shared, so Python allows a module the ability to "bring in" and use attributes from other modules to take advantage of work that has been done, maximizing code reusability. This process of associating attributes from other modules with your module is called importing. In a nutshell, modules are self-contained and organized pieces of Python code that can be shared.

If modules represent a logical way to organize your Python code, then files are a way to physically organize modules. To that end, each file is considered an individual module, and vice versa. The filename of a module is the module name appended with the .py file extension. There are several aspects we need to discuss with regard to what the file structure means to modules. Unlike other languages in which you import classes, in Python you import modules or module attributes.

Module Namespaces

We will discuss namespaces in detail later in this chapter, but the basic concept of a namespace is an individual set of mappings from names to objects. As you are no doubt aware, module names play an important part in the naming of their attributes. The name of the attribute is always prepended with the module name. For example, the atoi() function in the string module is called string.atoi(). Because only one module with a given name can be loaded into the Python interpreter, there is no intersection of names from different modules; hence, each module defines its own unique namespace. If I created a function called atoi() in my own module, perhaps mymodule, its name would be mymodule.atoi(). So even if there is a name conflict for an attribute, the fully qualified namereferring to an object via dotted attribute notationprevents an exact and conflicting match.

Search Path and Path Search

The process of importing a module requires a process called a path search. This is the procedure of checking "predefined areas" of the file system to look for your mymodule.py file in order to load the mymodule module. These predefined areas are no more than a set of directories that are part of your Python search path. To avoid the confusion between the two, think of a path search as the pursuit of a file through a set of directories, the search path.

There may be times where importing a module fails:

```
>>> import xxx
Traceback (innermost last):
File "<interactive input>", line 1, in ? ImportError: No module named xxx
```

When this error occurs, the interpreter is telling you it cannot access the requested module, and the likely reason is that the module you desire is not in the search path, leading to a path search failure.

A default search path is automatically defined either in the compilation or installation process. This search path may be modified in one of two places.

One is the PYTHONPATH environment variable set in the shell or command-line interpreter that invokes Python. The contents of this variable consist of a colon-delimited set of directory paths. If you want the interpreter to use the contents of this variable, make sure you set or update it before you start the interpreter or run a Python script.

Once the interpreter has started, you can access the path itself, which is stored in the sys module as the sys.path variable. Rather than a single string that is colon-delimited, the path has been "split" into a list of individual directory strings. Below is an example search path for a Unix machine. Your mileage willdefinitely vary as you go from system to system.

```
>>> sys.path
['', '/usr/local/lib/python2.x/', '/usr/local/lib/ python2.x/plat-sunos5',
'/usr/local/lib/python2.x/ lib-tk', '/usr/local/lib/python2.x/lib-dynload', '/
usr/local/lib/Python2.x/site-packages',]
```

Bearing in mind that this is just a list, we can definitely take liberty with it and modify it at our leisure. If you know of a module you want to import, yet its directory is not in the search path, by all means use the list's append() method to add it to the path, like so:

```
sys.path.append('/home/wesc/py/lib')
```

Once this is accomplished, you can then load your module. As long as one of the directories in the search path contains the file, then it will be imported. Of course, this adds the directory only to the end of your search path. If you want to add it elsewhere, such as in the beginning or middle, then you have to use the insert() list method for those. In our examples above, we are updating the sys.path attribute interactively, but it will work the same way if run as a script.

Here is what it would look like if we ran into this problem interactively:

```
>>> import sys
>>> import mymodule Traceback (innermost last):
File "<stdin>", line 1, in ?
ImportError: No module named mymodule
>>>
>>> sys.path.append('/home/wesc/py/lib')
>>> sys.path
['', '/usr/local/lib/python2.x/', '/usr/local/lib/ python2.x/plat-sunos5',
'/usr/local/lib/python2.x/
lib-tk', '/usr/local/lib/python2.x/lib-dynload', '/usr/ local/lib/python2.x/site-
packages','/home/wesc/py/lib']
>>>
>>> import mymodule
>>>
```

On the flip side, you may have too many copies of a module. In the case of duplicates, the interpreter will load the first module it finds with the given name while rummaging through the search path in sequential order.

To find out what modules have been successfully imported (and loaded) as well as from where, take a look at sys.modules. Unlike sys.path, which is a list of modules, sys.modules is a dictionary where the keys are the module names with their physical location as the values.

2.24 Namespaces

A namespace is a mapping of names (identifiers) to objects. The process of adding a name to a namespace consists of binding the identifier to the object (and increasing the reference count to the object by one). The Python Language Reference also includes the following definitions: "changing the mapping of a name is called rebinding [, and] removing a name is unbinding."

As briefly introduced in Chapter 11, there are either two or three active namespaces at any given time during execution. These three namespaces are the local, global, and built-ins namespaces, but local name-spaces come and go during execution, hence the "two or three" we just alluded to. The names
accessible from these namespaces are dependent on their loading order, or the order in which the namespaces are brought into the system.

The Python interpreter loads the built-ins namespace first. This consists of the names in thebuiltins module. Then the global namespace for the executing module is loaded, which then becomes the active namespace when the module begins execution. Thus we have our two active namespaces.

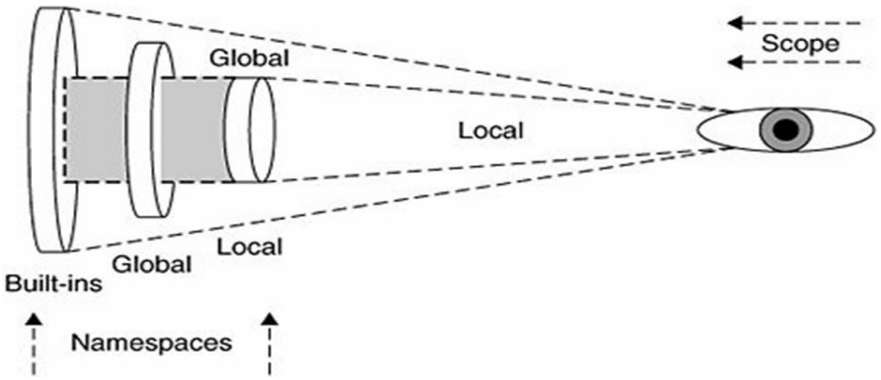

Figure:2.2: Namespaces versus variable scope

2.25 Importing Modules & Attributes

Importing a module requires the use of the import statement, whose syntax is:

```
import module1
import module2[
:
import moduleN
```

It is also possible to import multiple modules on the same line like this ...

```
import module1[, module2[,... moduleN]]
```

... but the resulting code is not as readable as having multiple import statements. Also, there is no performance hit and no change in the way that the Python bytecode is generated, so by all means, use the first form, which is the preferred form. When this statement is encountered by the interpreter, the module is imported if found in the search path. Scoping rules apply, so if imported from the top level of a module, it has global scope; if imported from a function, it has local scope. When a module is imported the first time, it is loaded and executed.

The from-import Statement

It is possible to import specific module elements into your own module. By this, we really mean importing specific names from the module into the current namespace. For this purpose, we can use the from-import statement, whose syntax is:

from module import name1[, name2[,... nameN]]

Multi-Line Import

The multi-line import feature was added in Python 2.4 specifically for long from-import statements. When importing many attributes from the same module, import lines of code tend to get long and wrap, requiring a NEWLINE-escaping backslash. Here is the example imported (pun intended) directly from PEP 328:

from Tkinter import Tk, Frame, Button, Entry, Canvas, \ Text, LEFT, DISABLED, NORMAL, RIDGE, END

Your other option is to have multiple from-import statements:

from Tkinter import Tk, Frame, Button, Entry, Canvas, Text
from Tkinter import LEFT, DISABLED, NORMAL, RIDGE, END

We are also trying to stem usage on the unfavored from Tkinter import * (see the Core Style sidebar in Section 12.5.3). Instead, programmers should be free to use Python's standard grouping mechanism (parentheses) to create a more reasonable multi-line import statement:

from Tkinter import (Tk, Frame, Button, Entry, Canvas, Text, LEFT, DISABLED, NORMAL, RIDGE, END)

You can find out more about multi-line imports in the documentation or in PEP 328.

Extended Import Statement (as)

There are times when you are importing either a module or module attribute with a name that you are already using in your application, or perhaps it is a name that you do not want to use. Maybe the name is too long to type everywhere, or more subjectively, perhaps it is a name that you just plain do not like.

This had been a fairly common request from Python programmers: the ability to import modules and module attributes into a program using names other than their original given names. One commonworkaround is to assign the module name to a variable:

```
>>> import longmodulename
>>> short = longmodulename
>>> del longmodulename
```

In the example above, rather than using longmodulename.attribute, you would use the short.attribute to access the same object. (A similar analogy can be made with importing module attributes using from- import, see below.) However, to do this over and over again and in multiple modules can be annoying and seem wasteful. Using extended import, you can change the locally bound name for what you are importing. Statements like ...

```
import Tkinter
from cgi import FieldStorage
```

. . . can be replaced by . . .

```
import Tkinter as tk
from cgi import FieldStorage as form
```

This feature was added in Python 2.0. At that time, "as" was not implemented as a keyword; it finally became one in Python 2.6. For more information on extended import, see the Python Language Reference Manual and PEP 221.

2.26 Module Built-in Functions

The importation of modules has some functional support from the system. We will look at those now.

import ()

The import () function is new as of Python 1.5, and it is the function that actually does the importing, meaning that the import statement invokes the import () function to do its work. The purpose of making this a function is to allow for overriding it if the user is inclined to develop his or her own importation algorithm.

The syntax of import () is:

import (module_name[, globals[, locals[, fromlist]]])

The module_name variable is the name of the module to import, globals is the dictionary of current names in the global symbol table, locals is the dictionary of current names in the local symbol table, and fromlist is a list of symbols to import the way they would be imported using the from-import statement.

The globals, locals, and fromlist arguments are optional, and if not provided, default to globals(), locals(), and [], respectively.

Calling import sys can be accomplished with

sys = import ('sys')

globals() and locals()

The globals() and locals() built-in functions return dictionaries of the global and local namespaces, respectively, of the caller. From within a function, the local namespace represents all names defined for execution of that function, which is what locals() will return. globals(), of course, will return those names globally accessible to that function.

From the global namespace, however, globals() and locals() return the same dictionary because the global namespace is as local as you can get while executing there. Here is a little snippet of code that calls both functions from both namespaces:

```
def foo():
print '\ncalling foo()...' aString = 'bar'
anInt = 42
print "foo()'s globals:", globals().keys()
print "foo()'s locals:", locals().keys()

print " main 's globals:", globals().keys()
print " main 's locals:", locals().keys()

foo()
```

We are going to ask for the dictionary keys only because the values are of no consequence here (plus they make the lines wrap even more in this text). Executing this script, we get the following output:

```
$ namespaces.py
  main 's globals: [' doc ', 'foo', ' name ',    ' builtins ']
  main 's locals: [' doc ', 'foo', ' name ',    ' builtins ']

calling foo()...
foo()'s globals: [' doc ', 'foo', ' name ',   ' builtins ']
foo()'s locals: ['anInt', 'aString']
```

reload()

The reload() built-in function performs another import on a previously imported module. The syntax of
reload() is:

```
reload(module)
```

module is the actual module you want to reload. There are some criteria for using the reload() module. The first is that the module must have been imported in full (not by using from-import), and it must have loaded successfully. The second rule follows from the first, and that is the argument to reload() the module itself and not a string containing the module name, i.e., it must be something like reload (sys) instead of reload('sys').

Also, code in a module is executed when it is imported, but only once. A second import does not re- execute the code, it just binds the module name. Thus reload() makes sense, as it overrides this default behavior.

2.27 Packages

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages. Packages were added to Python 1.5 to aid with a variety of problems including:

- Adding hierarchical organization to flat namespace
- Allowing developers to group related modules
- Allowing distributors to ship directories vs. bunch of files
- Helping resolve conflicting module names

Along with classes and modules, packages use the familiar attribute/dotted attribute notation to access their elements. Importing modules within packages use the standard import and from-import statements.

Directory Structure

For our package examples, we will assume the directory structure below:

```
Phone/
  init .py common_util.py Voicedta/
  init .py Pots.py Isdn.py
Fax/
  init .py G3.py
Mobile/
  init .py Analog.py Digital.py
  init .py Numeric.py
```

Phone is a top-level package and Voicedta, etc., are subpackages. Import subpackages by using import like this:

```
import Phone.Mobile.Analog Phone.Mobile.Analog.dial()
```

Alternatively, you can use from-import in a variety of ways:

The first way is importing just the top-level subpackage and referencing down the subpackage tree using the attribute/dotted notation:

```
from Phone import Mobile Mobile.Analog.dial('555-1212')
```

Furthermore, we can go down one more subpackage for referencing:

```
from Phone.Mobile import Analog Analog.dial('555-1212')
```

In fact, you can go all the way down in the subpackage tree structure:

```
from Phone.Mobile.Analog import dial dial('555-1212')
```

In our above directory structure hierarchy, we observe a number of init .py files. These are initializer modules that are required when using from-import to import subpackages but they can be empty if not used. Quite often, developers forget to add _inti_.py files to their package directories, so starting in Python 2.5, this triggers an ImportWarning message.

However, it is silently ignored unless the -Wd option is given when launching the interpreter.

Using from-import with Packages

Packages also support the from-import all statement:

```
from package.module import *
```

However, such a statement is dependent on the operating system's filesystem for Python to determine which files to import. Thus the all variable in init .py is required. This variable contains all the module names that should be imported when the above statement is invoked if there is such a thing. It consists of a list of module names as strings.

Absolute Import

As the use of packages becomes more pervasive, there have been more cases of the import of subpackages that end up clashing with (and hiding or shadowing) "real" or standard library modules (actually their names). Package modules will hide any equivalently-named standard library module because it will look inside the package first to perform a relative import, thus hiding access to the standard library module.

Because of this, all imports are now classified as absolute, meaning that names must be packages or modules accessible via the Python path (sys.path or PYTHONPATH).

The rationale behind this decision is that subpackages can still be accessed via sys.path, i.e., import Phone.Mobile.Analog. Prior to this change, it was legal to have just import Analog from modules inside the Mobile subpackage.

As a compromise, Python allows relative importing where programmers can indicate the location of a sub-package to be imported by using leader dots in front of the module or package name. The absolute import feature is the default starting in Python 2.7. (This feature, absolute_import, can be imported from future starting in version 2.5.) You can read more about absolute import in PEP 328.

Relative Import

As described previously, the absolute import feature takes away certain privileges of the module writer of packages. With this loss of freedom in import statements, something must be made available to proxy for that loss. This is where a relative import comes in. The relative import feature alters the import syntax slightly to let programmers tell the importer where to find a module in a subpackage. Because the import statements are always absolute, relative imports only apply to from-import statements.

The first part of the syntax is a leader dot to indicate a relative import. From there, any additional dot represents a single level above the current from where to start looking for the modules being imported.

Let us look at our example above again. From within Analog.Mobile. Digital, i.e., the Digital.py module, we cannot simply use this syntax anymore. The following will either still work in older versions of Python, generate a warning, or will not work in more contemporary versions of Python:

```
import Analog
from Analog import dial
```

This is due to the absolute import limitation. You have to use either the absolute or relative imports. Below are some valid imports:

```
from Phone.Mobile.Analog import dial
from .Analog import dial
from ..common_util import setup
from ..Fax import G3.dial.
```

Relative imports can be used starting in Python 2.5. In Python 2.6, a deprecation warning will appear for all intra-package imports not using the relative import syntax. You can read more about relative import in the Python documentation and in PEP 328.

2.28 Other Features of Modules

Auto-Loaded Modules

When the Python interpreter starts up in standard mode, some modules are loaded by the interpreter for system use. The only one that affects you is the builtin module, which normally gets loaded in as the builtins module.

The sys.modules variable consists of a dictionary of modules that the interpreter has currently loaded (in full and successfully) into the interpreter. The module names are the keys, and the location from which they were imported are the values.

For example, in Windows, the sys.modules variable contains a large number of loaded modules, so we will shorten the list by requesting only the module names. This is accomplished by using the dictionary's keys() method:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'exceptions', ' main ', 'ntpath',
'strop', 'nt', 'sys',          ' builtin ', 'site',
'signal', 'UserDict', 'string', 'stat']
```

The loaded modules for Unix are quite similar:

```
>>> import sys
>>> sys.modules.keys()
['os.path', 'os', 'readline', 'exceptions',
' main ', 'posix', 'sys', ' builtin ', 'site',
'signal', 'UserDict', 'posixpath', 'stat']
```

Preventing Attribute Import

If you do not want module attributes imported when a module is imported with "from module import *", prepend an underscore (_) to those attribute names (you do not want imported). This minimal level of data hiding does not apply if the entire module is imported or if you explicitly import a "hidden" attribute, e.g., import foo._bar.

Case-Insensitive Import

There are various operating systems with case-insensitive file systems. Prior to version 2.1, Python attempted to "do the right thing" when importing modules on the various supported platforms, but with the growing popularity of the MacOS X and Cygwin platforms, certain deficiencies could no longer be ignored, and support needed to be cleaned up.

The world was pretty clean-cut when it was just Unix (case-sensitive) and Win32 (case-insensitive), but these new case-insensitive systems coming online were not ported with the case-insensitive features.
PEP 235, which specifies this feature, attempts to address this weakness as well as taking away some "hacks" that had existed for other systems to make importing modules more consistent.

The bottom line is that for case-insensitive imports to work properly, an environment variable named PYTHONCASEOK must be defined. Python will then import the first module name that is found (in a case- insensitive manner) that matches. Otherwise Python will perform its native case-sensitive module name matching and import the first matching one it finds.

Source Code Encoding

Starting in Python 2.3, it is now possible to create your Python module file in a native encoding other than 7-bit ASCII. Of course ASCII is the default, but with an additional encoding directive at the top of your Python modules, it will enable the importer to parse your modules using the specified encoding anddesignate natively encoded Unicode strings correctly so you do not have to worry about editing your source files in a plain ASCII text editor and have to individually "Unicode-tag" each string literal.

An example directive specifying a UTF-8 file can be declared like this:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

If you execute or import modules that contain non-ASCII Unicode string literals and do not have an encoding directive at the top, this will result in a DeprecationWarning in Python 2.3 and a syntax error starting in 2.5. You can read more about source code encoding in PEP 263.

Import Cycles

Working with Python in real-life situations, you discover that it is possible to have import loops. If you have ever worked on any large Python project, you are likely to have run into this situation.

Let us take a look at an example. Assume we have a very large product with a very complex command- line interface (CLI). There are a million commands for your product, and as a result, you have an overly massive handler (OMH) set. Every time a new feature is added, from one to three new commands must be added to support the new feature. This will be our omh4cli.py script:

```
from cli4vof import cli4vof
# command line interface utility function
def cli_util():
pass

# overly massive handlers for the command line interface
def omh4cli():
:
cli4vof()
:
omh4cli()
```

You can pretend that the (empty) utility function is a very popular piece of code that most handlers must use. The overly massive handlers for the command-line interface are all in the omh4cli() function. If we have to add a new command, it would be called from here.

Now, as this module grows in a boundless fashion, certain smarter engineers decide to split off their new commands into a separate module and just provide hooks in the original module to access the new stuff. Therefore, the code is easier to maintain, and if bugs were found in the new stuff, one would not have to search through a one-megabyte-plus-sized Python file.

In our case, we have an excited product manager asking us to add a "very outstanding feature" (VOF). Instead of integrating our stuff into omh4cli.py, we create a new script, cli4vof.py:

```
import omh4cli
# command-line interface for a very outstanding feature
def cli4vof(): omh4cli.cli_util()
```

As mentioned before, the utility function is a must for every command, and because we do not want to cut and paste its code from the main handler, we import the main module and call it that way. To finish off our integration, we add a call to our handler into the main overly massive handler, omh4cli().

The problem occurs when the main handler omh4cli imports our new little module cli4vof (to get the new command function) because cli4vof imports omh4cli (to get the utility function). Our module import fails because Python is trying to import a module that was not previously fully imported the first time:

```
$ python omh4cli.py
Traceback (most recent call last): File "omh4cli.py", line 3, in ?
from cli4vof import cli4vof
File "/usr/prod/cli4vof.py", line 3, in ? import omh4cli
File "/usr/prod/omh4cli.py", line 3, in ? from cli4vof import cli4vof
ImportError: cannot import name cli4vof
```

Notice the circular import of cli4vof in the traceback. The problem is that in order to call the utility function, cli4vof has to import omh4cli. If it did not have to do that, then omh4cli would have completed its import of cli4vof successfully and there would be no problem. The issue is that when omh4cli is attempting to import cli4vof, cli4vof is trying to import omh4cli. No one finishes an import, hence the error. This is just one example of an import cycle. There are much more complicated ones out in the real world.

The workaround for this problem is almost always to move one of the import statements, e.g., the offending one. You will commonly see import statements at the bottom of modules. As a beginning Python programmer, you are used to seeing them in the beginning, but if you ever run across import statements at the end of modules, you will now know why. In our case, we cannot move the import of omh4cli to the end, because if cli4vof() is called, it will not have the omh4cli name loaded yet:

```
$ python omh4cli.py
Traceback (most recent call last): File "omh4cli.py", line 3, in ?
from cli4vof import cli4vof
File "/usr/prod/cli4vof.py", line 7, in ? import omh4cli
File "/usr/prod/omh4cli.py", line 13, in ? omh4cli()
File "/usr/prod/omh4cli.py", line 11, in omh4cli cli4vof()
File "/usr/prod/cli4vof.py", line 5, in cli4vof omh4cli.cli_util()
NameError: global name 'omh4cli' is not defined
```

No, our solution here is to just move the import statement into the cli4vof() function declaration:

```
def cli4vof():
import omh4cli omh4cli.cli_util()
```

This way, the import of the cli4vof module from omh4cli completes successfully, and on the tail end, calling the utility function is successful because the omh4cli name is imported before it is called. As far as execution goes, the only difference is that from cli4vof, the import of omh4cli is performed when cli4vof.cli4vof() is called and not when the cli4vof module is imported.

Module Execution

There are many ways to execute a Python module: script invocation via the command-line or shell, execfile(), module import, interpreter -m option, etc.

UNIT III – REGULAR EXPRESSIONS & MULTITHREAD PROGRAMMING

3.1 Regular Expressions- Introduction

Manipulating text/data is a big thing. If you don't believe me, look very carefully at what computers primarily do today. Word processing, "fill-out-form" Web pages, streams of information coming from a database dump, stock quote information, news feedsthe list goes on and on. Because we may not know the exact text or data that we have programmed our machines to process, it becomes advantageous to be able to express this text or data in patterns that a machine can recognize and take action upon.

If I were running an electronic mail (e-mail) archiving company, and you were one of my customers who requested all his or her e-mail sent and received last February, for example, it would be nice if I could set a computer program to collate and forward that information to you, rather than having a human being read through your e-mail and process your request manually. You would be horrified (and infuriated) that someone would be rummaging through your messages, even if his or her eyes were supposed to be looking only at time-stamp. Another example request might be to look for a subject line like "ILOVEYOU" indicating a virus-infected message and remove those e-mail messages from your personal archive. So this begs the question of how we can program machines with the ability to look for patterns in text.

First Regular Expression

As we mentioned above, REs are strings containing text and special characters that describe a patternwith which to recognize multiple strings. We also briefly discussed a regular expression alphabet and for general text, the alphabet used for regular expressions is the set of all uppercase and lowercase letters plus numeric digits. Specialized alphabets are also possible, for instance, one consisting of only the characters "0" and "1". The set of all strings over this alphabet describes all binary strings, i.e., "0," "1," "00," "01," "10," "11," "100," etc.

Let us look at the most basic of regular expressions now to show you that although REs are sometimes considered an "advanced topic," they can also be rather simplistic. Using the standard alphabet for general text, we present some simple REs and the strings that their patterns describe. The following regular expressions are the most basic, "true vanilla," as it were. They simply consist of a string pattern that matches only one string, the string defined by the regular expression. We now present the REs followed by the strings that match them:

RE Pattern	**String(s) Matched**
foo	foo
Python	Python
abc123	abc123

The first regular expression pattern from the above chart is "foo." This pattern has no special symbols to match any other symbol other than those described, so the only string that matches this pattern is the string "foo." The same thing applies to "Python" and "abc123." The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition. It is these special symbols that allow an RE to match a set of strings rather than a single one.

3.2 Special Symbols and Characters

We will now introduce the most popular of the metacharacters, special characters and symbols, which give regular expressions their power and flexibility.

Matching More Than One RE Pattern with Alternation (|)

The pipe symbol (|), a vertical bar on your keyboard, indicates an alternation operation, meaning that it is used to choose from one of the different regular expressions, which are separated by the pipe symbol. For example, below are some patterns that employ alternation, along with the strings they match:

RE Pattern	Strings Matched
at|home	at, home
r2d2|c3po	r2d2, c3po
bat|bet|bit	bat, bet, bit

With this one symbol, we have just increased the flexibility of our regular expressions, enabling the matching of more than just one string. Alternation is also sometimes called union or logical OR.

Matching Any Single Character (.)

The dot or period (.) symbol matches any single character except for NEWLINE (Python REs have a compilation flag [S or DOTALL], which can override this to include NEWLINEs.). Whether letter, number, whitespace not including "\n," printable, non-printable, or a symbol, the dot can match them all.

RE Pattern	Strings Matched
f.o	Any character between "f" and "o", e.g., fao, f9o, f#o, etc.
..	Any pair of characters
.end	Any character before the string end

Matching from the Beginning or End of Strings or Word Boundaries (^/$ /\b /\B)

There are also symbols and related special characters to specify searching for patterns at the beginning and ending of strings. To match a pattern starting from the beginning, you must use the carat symbol
(^) or the special character \A (backslash-capital "A"). The latter is primarily for keyboards that do not have the carat symbol, i.e., international. Similarly, the dollar sign ($) or \Z will match a pattern from the end of a string.

Patterns that use these symbols differ from most of the others we describe in this chapter since they dictate location or position. In the Core Note above, we noted that a distinction is made between "matching," attempting matches of entire strings starting at the beginning, and "searching," attempting matches from anywhere within a string. With that said, here are some examples of "edge-bound" RE search patterns:

RE Pattern	Strings Matched
^From	Any string that starts with From
/bin/tcsh$	Any string that ends with /bin/tcsh
^Subject: hi$	Any string consisting solely of the string Subject: hi

Again, if you want to match either (or both) of these characters verbatim, you must use an escaping backslash. For example, if you wanted to match any string that ended with a dollar sign, one possible RE solution would be the pattern ".*\$$".

The \b and \B special characters pertain to word boundary matches. The difference between them is that\b will match a pattern to a word boundary, meaning that a pattern must be at the beginning of a word, whether there are any characters in front of it (word in the middle of a string) or not (word at the beginning of a line). And likewise, \B will match a pattern only if it appears starting in the middle of a word (i.e., not at a word boundary). Here are some examples:

RE Pattern	**Strings Matched**
the	Any string containing the
\bthe	Any word that starts with the
\bthe\b	Matches only the word the
\Bthe	Any string that contains but does not begin with the

3.3 REs and Python

Now that we know all about regular expressions, we can examine how Python currently supports regular expressions through the re module. The re module was introduced to Python in version 1.5. If you are using an older version of Python, you will have to use the now-obsolete regex and regsub modulesthese older modules are more Emacs-flavored, are not as full-featured, and are in many ways incompatible with the current re module. Both modules were removed from Python in 2.5, and import either of the modules from 2.5 and above triggers Import Error exception.

However, regular expressions are still regular expressions, so most of the basic concepts from this section can be used with the old regex and regsub software. In contrast, the new re module supports the more powerful and regular Perl-style (Perl5) REs, allows multiple threads to share the same compiled RE objects, and supports named subgroups. In addition, there is a transition module called reconvert to help developers move from regex/regsub to re. However, be aware that although there are different flavors of regular expressions, we will primarily focus on the current incarnation for Python.

The re engine was rewritten in 1.6 for performance enhancements as well as adding Unicode support. The interface was not changed, hence the reason the module name was left alone. The new re engineknown internally as sre thus replaces the existing 1.5 engineinternally called pcre.

re Module: Core Functions and Methods

The chart in Table 15.2 lists the more popular functions and methods from the re module. Many of these functions are also available as methods of compiled regular expression objects "regex objects" and RE "match objects." In this subsection, we will look at the two main functions/methods, match() and search (), as well as the compile() function. We will introduce several more in the next section, but for more information on all these and the others that we do not cover, we refer you to the Python documentation.

3.4 Multithreaded Programming – Introduction

Before the advent of multithreaded (MT) programming, running of computer programs consisted of a single sequence of steps that were executed in synchronous order by the host's central processing unit (CPU). This style of execution was the norm whether the task itself required the sequential ordering of steps or if the entire program was actually an aggregation of multiple subtasks. What if these subtasks were independent, having no causal relationship (meaning that results of subtasks do not affect other subtask outcomes)? Is it not logical, then, to want to run these independent tasks all at the same time? Such parallel processing could significantly improve the performance of the overall task. This is what MT programming is all about.

MT programming is ideal for programming tasks that are asynchronous in nature, require multiple concurrent activities, and where the processing of each activity may be nondeterministic, i.e., random and unpredictable. Such programming tasks can be organized or partitioned into multiple streams of execution where each has a specific task to accomplish. Depending on the application, these subtasks may calculate intermediate results that could be merged into a final piece of output.

While CPU-bound tasks may be fairly straightforward to divide into subtasks and executed sequentially or in a multithreaded manner, the task of managing a single-threaded process with multiple external sources of input is not as trivial. To achieve such a programming task without multithreading, a sequential program must use one or more timers and implement a multiplexing scheme.

A sequential program will need to sample each I/O (input/output) terminal channel to check for user input; however, it is important that the program does not block when reading the I/O terminal channel because the arrival of user input is nondeterministic, and blocking would prevent processing of other I/O channels. The sequential program must use non-blocked I/O or blocked I/O with a timer (so that blocking is only temporary).

Because the sequential program is a single thread of execution, it must juggle the multiple tasks that it needs to perform, making sure that it does not spend too much time on any one task, and it must ensure that user response time is appropriately distributed. The use of a sequential program for this type of task often results in a complicated flow of control that is difficult to understand and maintain.

Using an MT program with a shared data structure such as a Queue (a multithreaded queue data structure discussed later in this chapter), this programming task can be organized with a few threads that have specific functions to perform:

- UserRequestThread: Responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.
- RequestProcessor: A thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.
- ReplyThread: Responsible for taking output destined for the user and either sending it back, if in
- a networked application, or writing data to the local file system or database.

Organizing this programming task with multiple threads reduces the complexity of the program and enables an implementation that is clean, efficient, and well organized. The logic in each thread is typically less complex because it has a specific job to do. For example, the UserRequestThread simply reads input from a user and places the data into a queue for further processing by another thread, etc. Each thread has its own job to do; you merely have to design each type of thread to do one thing and do it well. Use of threads for specific tasks is not unlike Henry Ford's assembly line model for manufacturing automobiles.

3.5 Threads and Processes

Processes

Computer programs are merely executable, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A process (sometimes called a heavyweight process) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes.

Processes can also fork or spawn new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

Threads:

Threads (sometimes called lightweight processes) are similar to processes except that they all execute within the same process, and thus all share the same context. They can be thought of as "mini- processes" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be preempted (interrupted) and temporarily put on hold (also known as sleeping) while other threads are running this is called yielding.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with one another more easily than if they were separate processes. Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial "back of the line" to await more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a race condition. Fortunately, most thread libraries come with some sort of synchronization primitives that allow the thread manager to control execution and access.

Another caveat is that threads may not be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.

3.6 Python, Threads, and the Global Interpreter Lock

Global Interpreter Lock (GIL)

Execution of Python code is controlled by the Python Virtual Machine (aka the interpreter main loop). Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only one is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

1. Set the GIL
2. Switch in a thread to run
3. Execute either ...
    a.For a specified number of bytecode instructions, or
    b. If the thread voluntarily yields control (can be accomplished time.sleep(0))
4. Put the thread back to sleep (switch out thread)
5. Unlock the GIL, and ...
6. Do it all over again (lather, rinse, repeat)

When a call is made to external code, i.e., any C/C++ extension built-in function, the GIL will be locked until it has completed (since there are no Python bytecodes to count as the interval). Extension programmers do have the ability to unlock the GIL, however, so you being the Python developer shouldn't have to worry about your Python code locking up in those situations.

As an example, for any Python I/O-oriented routines (which invoke built-in operating system C code), the GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed. Code that doesn't have much I/O will tend to keep the processor (and GIL) for the full interval a thread is allowed before it yields. In other words, I/O-bound Python programs stand a much better chance of being able to take advantage of a multithreaded environment than CPU-bound code.

3.7 Thread Module

Let's take a look at what the tHRead module has to offer. In addition to being able to spawn threads, the tHRead module also provides a basic synchronization data structure called a lock object (aka primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore). As we mentioned earlier, such synchronization primitives go hand in hand with thread management.

Using the thread Module (mtsleep1.py)

The same loops from onethr.py are executed, but this time using the simple multithreaded mechanism provided by the thread module. The two loops are executed concurrently (with the shorter one finishing first, obviously), and the total elapsed time is only as long as the slowest thread rather than the total time for each separately.

```
import thread
from time import sleep, ctime

def loop0():
print 'start loop 0 at:', ctime() sleep(4)
print 'loop 0 done at:', ctime()

def loop1():
print 'start loop 1 at:', ctime() sleep(2)
print 'loop 1 done at:', ctime()

def main():
print 'starting at:', ctime() thread.start_new_thread(loop0, ())
thread.start_new_thread(loop1, ()) sleep(6)
print 'all DONE at:', ctime()

if name == ' main ': main()
```

start_new_thread() requires the first two arguments, so that is the reason for passing in an empty tuple even if the executing function requires no arguments.

Upon execution of this program, our output changes drastically. Rather than taking a full 6 or 7 seconds, our script now runs in 4, the length of time of our longest loop, plus any overhead.

```
$ mtsleep1.py
starting at: Sun Aug 13 05:04:50 2006
start loop 0 at: Sun Aug 13 05:04:50 2006
start loop 1 at: Sun Aug 13 05:04:50 2006
loop 1 done at: Sun Aug 13 05:04:52 2006
loop 0 done at: Sun Aug 13 05:04:54 2006
all DONE at: Sun Aug 13 05:04:56 2006
```

The pieces of code that sleep for 4 and 2 seconds now occur concurrently, contributing to the lower overall runtime. You can even see how loop 1 finishes before loop 0.

The only other major change to our application is the addition of the "sleep(6)" call. Why is this necessary? The reason is that if we did not stop the main thread from continuing, it would proceed to the next statement, displaying "all done" and exit, killing both threads running loop0() and loop1().

We did not have any code that told the main thread to wait for the child threads to complete before continuing. This is what we mean by threads requiring some sort of synchronization. In our case, we used another sleep() call as our synchronization mechanism. We used a value of 6 seconds because we know that both threads (which take 4 and 2 seconds, as you know) should have completed by the time the main thread has counted to 6.
You are probably thinking that there should be a better way of managing threads than creating that extra delay of 6 seconds in the main thread. Because of this delay, the overall runtime is no better than in our single-threaded version.

Using sleep() for thread synchronization as we did is not reliable. What if our loops had independent and varying execution times? We may be exiting the main thread too early or too late. This is where locks come in.

Making yet another update to our code to include locks as well as getting rid of separate loop functions, we get mtsleep2.py, presented in Example 18.3. Running it, we see that the output is similar to mtsleep1. py. The only difference is that we did not have to wait the extra time for mtsleep1.py to conclude. By using locks, we were able to exit as soon as both threads had completed execution.

```
$ mtsleep2.py
starting at: Sun Aug 13 16:34:41 2006
start loop 0 at: Sun Aug 13 16:34:41 2006
start loop 1 at: Sun Aug 13 16:34:41 2006
loop 1 done at: Sun Aug 13 16:34:43 2006
loop 0 done at: Sun Aug 13 16:34:45 2006
all DONE at: Sun Aug 13 16:34:45 2006
```

3.8 Threading Module

We will now introduce the higher-level tHReading module, which gives you not only a THRead class but also a wide variety of synchronization mechanisms to use to your heart's content.

Using the tHReading Module (mtsleep3.py)

The Thread class from the threading module has a join() method that lets the main thread wait for thread completion.

```
1   #!/usr/bin/env python
2
3   import threading
4   from time import sleep, ctime
5
6   loops = [4,2]
7
8   def loop(nloop, nsec):
9   print 'start loop', nloop, 'at:', ctime()
10  sleep(nsec)
11  print 'loop', nloop, 'done at:', ctime()
12
13  def main():
14  print 'starting at:', ctime()
15  threads = []
16  nloops = range(len(loops))
17
18  for i in nloops:
19  t = threading.Thread(target=loop,
20  args=(i, loops[i]))
21  threads.append(t)
22
23  for i in nloops:# start threads
24  threads[i].start()
25
```

```
26      for i in nloops:# wait for all
27      threads[i].join()       # threads to finish
28
29      print ‘all DONE at:’, ctime()
30
31      if_name == ‘_main_’:
32              main()
```

When we run it, we see output similar to its predecessors' output:

```
$ mtsleep3.py
starting at: Sun Aug 13 18:16:38 2006
start loop 0 at: Sun Aug 13 18:16:38 2006
start loop 1 at: Sun Aug 13 18:16:38 2006
loop 1 done at: Sun Aug 13 18:16:40 2006
loop 0 done at: Sun Aug 13 18:16:42 2006
all DONE at: Sun Aug 13 18:16:42 2006
```

So what did change? Gone are the locks that we had to implement when using the tHRead module. Instead, we create a set of Thread objects. When each Thread is instantiated, we dutifully pass in the function (target) and arguments (args) and receive a THRead instance in return. The biggest difference between instantiating Thread [calling Thread()] and invoking thread.start_new_thread() is that the new thread does not begin execution right away. This is a useful synchronization feature, especially when you don't want the threads to start immediately.

Once all the threads have been allocated, we let them go off to the races by invoking each thread's start () method, but not a moment before that. And rather than having to manage a set of locks (allocating, acquiring, releasing, checking lock state, etc.), we simply call the join() method for each thread. join() will wait until a thread terminates, or, if provided, a timeout occurs. Use of join() appears much cleaner than an infinite loop waiting for locks to be released (causing these locks to sometimes be known as "spin locks").

One other important aspect of join() is that it does not need to be called at all. Once threads are started, they will execute until their given function completes, whereby they will exit. If your main thread has things to do other than wait for threads to complete (such as other processing or waiting for new client requests), it should by all means do so. join() is useful only when you want to wait for thread completion.

3.9 Related Modules

The table below lists some of the modules you may use when programming multithreaded applications.

Threading-Related Standard Library Modules

Module	**Description**
tHRead	Basic, lower-level thread module
threading	Higher-level threading and synchronization objects Queue
Synchronized	FIFO queue for multiple threads mutex Mutual exclusion objects
SocketServer	TCP and UDP managers with some threading control

UNIT IV – GUI & WEB PROGRAMMING

4.1 GUI Programming – Introduction

Tkinter is Python's default GUI library. It is based on the Tk toolkit, originally designed for the Tool Command Language (Tcl). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter). With the GUI development portability and flexibility of Tk, along with the simplicity of scripting language integrated with the power of systems language, you are given the tools to rapidly design and implement a wide variety of commercial-quality GUI applications.

If you are new to GUI programming, you will be pleasantly surprised at how easy it is. You will also find that Python, along with Tkinter, provides a fast and exciting way to build applications that are fun (and perhaps useful) and that would have taken much longer if you had had to program directly in C/C++ with the native windowing system's libraries. Once you have designed the application and the look and feel that goes along with your program, you will use basic building blocks known as widgets to piece together the desired components, and finally, to attach functionality to "make it real."

If you are an old hand at using Tk, either with Tcl or Perl, you will find Python a refreshing way to program GUIs. On top of that, it provides an even faster rapid prototyping system for building them. Remember that you also have Python's system accessibility, networking functionality, XML, numerical and visual processing, database access, and all the other standard library and third-party extension modules.

Once you get Tkinter up on your system, it will take less than 15 minutes to get your first GUI application running.

Getting Tkinter Installed and Working

Like threading, Tkinter is not necessarily turned on by default on your system. You can tell whether Tkinter is available for your Python interpreter by attempting to import the Tkinter module. If Tkinter is available, then no errors occur:

```
>>> import Tkinter
>>>
```

If your Python interpreter was not compiled with Tkinter enabled, the module import fails:

```
>>> import Tkinter Traceback (innermost last):
File "<stdin>", line 1, in ?
File "/usr/lib/python1.5/lib-tk/Tkinter.py", line 8, in ?
import _tkinter # If     this fails your Python may not be configured for Tk
ImportError: No module named _tkinter
```

You may have to recompile your Python interpreter to get access to Tkinter. This usually involves editing the Modules/Setup file and enabling all the correct settings to compile your Python interpreter with hooks to Tkinter or choosing to have Tk installed on your system. Check the README file for your Pythondistribution for specific instructions on getting Tkinter to compile on your system. Be sure, after your compilation, that you start the new Python interpreter you just created; otherwise, it will act just like your old one without Tkinter (and in fact, it is your old one).

Client/Server ArchitectureTake Two

In the earlier chapter on network programming, we introduced the notion of client/server computing. A windowing system is another example of a software server. These run on a machine with an attached display, such as a monitor of some sort. There are clients, tooprograms that require a windowing environment to execute, also known as GUI applications. Such applications cannot run without a windows system.

The architecture becomes even more interesting when networking comes into play. Usually when a GUI application is executed, it displays to the machine that it started on (via the windowing server), but it is possible in some networked windowing environments, such as the X Window system on Unix, to choose another machine's window server to display to. In such situations, you can be running a GUI program on one machine, but have it displayed on another!

4.2 Tkinter and Python Programming

Tkinter Module: Adding Tk to your Applications

So what do you need to do to have Tkinter as part of your application? Well, first of all, it is not necessary to have an application already. You can create a pure GUI if you want, but it probably isn't too useful without some underlying software that does something interesting.

There are basically five main steps that are required to get your GUI up and running:

1. Import the Tkinter module (or from Tkinter import *).
2. Create a top-level windowing object that contains your entire GUI application.
3. Build all your GUI components (and functionality) on top (or "inside") of your top-level windowing object.
4. Connect these GUI components to the underlying application code.
5. Enter the main event loop.

Introduction to GUI Programming

Before going to the examples, we will give you a brief introduction to GUI application development in general. This will provide you with some of the background you need to move forward.

Setting up a GUI application is similar to an artist's producing a painting. Conventionally, there is a single canvas onto which the artist must put all the work. The way it works is like this: You start with a clean slate, a "top-level" windowing object on which you build the rest of your components. Think of it as a foundation to a house or the easel for an artist. In other words, you have to pour the concrete or set up your easel before putting together the actual structure or canvas on top of it. In Tkinter, this foundation is known as the top-level window object.

In GUI programming, a top-level root windowing object contains all of the little windowing objects that will be part of your complete GUI application. These can be text labels, buttons, list boxes, etc. These individual little GUI components are known as widgets. So when we say create a top-level window, we just mean that you need such a thing as a place where you put all your widgets. In Python, this would typically look like this line:

```
top = Tkinter.Tk() # or just Tk() with "from Tkinter import *"
```

The object returned by Tkinter.Tk() is usually referred to as the root window, hence the reason whysome applications use root rather than top to indicate as such. Top-level windows are those that show up standalone as part of your application. You may have more than one top-level window for your GUI, but only one of them should be your root window. You may choose to completely design all your widgets first, then add the real functionality, or do a little of this and a little of that along the way. (This means mixing and matching steps 3 and 4 from our list.)

Widgets may be standalone or be containers. If a widget "contains" other widgets, it is considered the parent of those widgets. Accordingly, if a widget is "contained" in another widget, it's considered a child of the parent, the parent being the next immediate enclosing container widget.

Usually, widgets have some associated behaviors, such as when a button is pressed, or text is filled into a text field. These types of user behaviors are called events, and the actions that the GUI takes to respond to such events are known as callbacks.

Actions may include the actual button press (and release), mouse movement, hitting the RETURN or Enter key, etc. All of these are known to the system literally as events. The entire system of events that occurs from the beginning to the end of a GUI application is what drives it. This is known as event- driven processing.

One example of an event with a callback is a simple mouse move. Let's say the mouse pointer is sitting somewhere on top of your GUI application. If the mouse is moved to another part of your application, something has to cause the movement of the mouse on your screen so that it looks as if it is moving to another location. These are mouse move events that the system must process to give you the illusion (and reality) that your mouse is moving across the window. When you release the mouse, there are no more events to process, so everything just sits there quietly on the screen again.

The event-driven processing nature of GUIs fits right in with client/server architecture. When you start a GUI application, it must perform some setup procedures to prepare for the core execution, just as when a network server has to allocate a socket and bind it to a local address. The GUI application must establish all the GUI components, then draw (aka render or paint) them to the screen. Tk has a couple of geometry managers that help position the widget in the right place; the main one that you will use is called Pack, aka the packer. Another geometry manager is Gridthis is where you specify GUI widgets to be placed in grid coordinates, and Grid will render each object in the GUI in their grid position. For us, we will stick with the packer.

Once the packer has determined the sizes and alignments of all your widgets, it will then place them on the screen for you. When all of the widgets, including the top-level window, finally appear on your screen, your GUI application then enters a "server-like" infinite loop. This infinite loop involves waiting for a GUI event, processing it, then going back to wait for the next event.

The final step we described above says to enter the main loop once all the widgets are ready. This is the "server" infinite loop we have been referring to. In Tkinter, the code that does this is:

```
Tkinter.mainloop()
```

This is normally the last piece of sequential code your program runs. When the main loop is entered, the GUI takes over execution from there. All other action is via callbacks, even exiting your application.
When you pull down the File menu to click on the Exit menu option or close the window directly, a callback must be invoked to end your GUI application.

Top-Level Window: Tkinter.Tk()

We mentioned above that all main widgets are built into the top-level window object. This object iscreated by the Tk class in Tkinter and is created via the normal instantiation:

```
>>> import Tkinter
>>> top = Tkinter.Tk()
```

4.3 Brief Tour of other GUIs

We hope to eventually develop an independent chapter on general GUI development using many of the abundant number of graphical toolkits that exist under Python, but alas, that is for the future. As a proxy, we would like to present a single simple GUI application written using four of the more popular and available toolkits out there: Tix (Tk Interface eXtensions), Pmw (Python MegaWidgets Tkinter extension), wxPython (Python binding to wxWidgets), and PyGTK (Python binding to GTK+). Links to where you can get more information and/or download these toolkits can be found in the reference section at the end of this chapter.

The Tix module is already available in the Python standard library. You must download the others, which are third party. Since Pmw is just an extension to Tkinter, it is the easiest to install (just extract into your site packages). wxPython and PyGTK involve the download of more than one file and building (unless you opt for the Win32 versions where binaries are usually available). Once the toolkits are installed and verified, we can begin. Rather than just sticking with the widgets we've already seen in this chapter, we'd like to introduce a few more complex widgets for these examples.

In addition to the Label and Button widgets we have seen before, we would like to introduce the Control or SpinButton and ComboBox. The Control widget is a combination of a text widget with a value inside being "controlled" or "spun up or down" by a set of arrow buttons close by, and the ComboBox is usually a text widget and a pulldown menu of options where the currently active or selected item in the list is displayed in the text widget.

Our application is fairly basic: pairs of animals are being moved around, and the number of total animals can range from a pair to a dozen max. The Control is used to keep track of the total number while the ComboBox is a menu containing the various types of animals that can be selected. In Figure, each image shows the state of the GUI application immediately after launching. Note that the default number of animals is two, and no animal type has been selected yet.

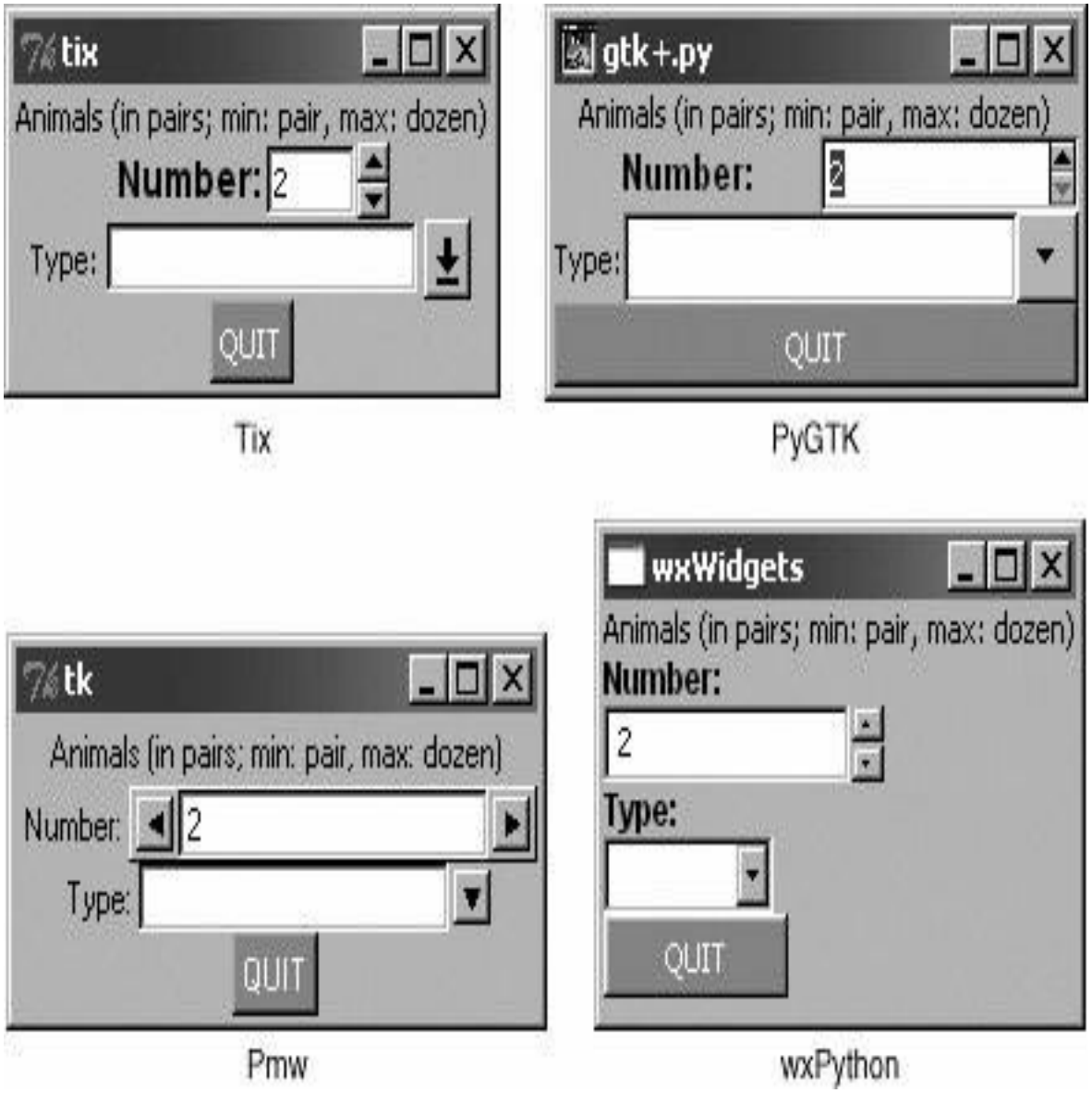

Fig: 4.1 Application using various GUIs under Win32

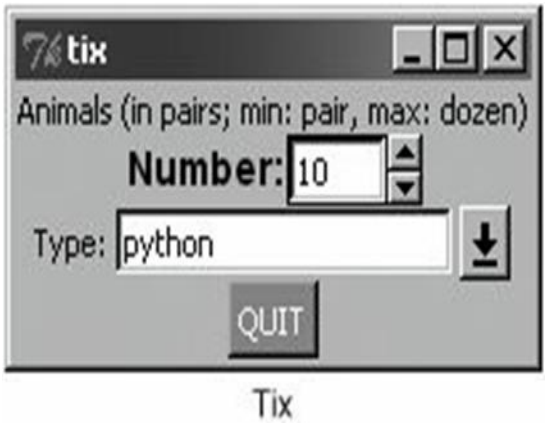

Fig: 4.2 After modifying the Tix GUI version of our application (animalTix. pyw)

Tix GUI Demo (animalTix.pyw)

Our first example uses the `Tix` *module. Tix comes with Python!*

```
1  #!/usr/bin/envpyth
2
3  from Tkinter import Label, Button, END
4  from Tix import Tk, Control, ComboBox
5
6  top = Tk()
7  top.tk.eval('package require Tix')
8
9  lb = Label(top,
11
lb.pack
() 12
13 ct = Control(top,label='Number:',
14    integer=True, max=12, min=2,
      value=2,step=2)
15 ct.label.config(font='Helvetica -14bold')
16 ct
.pack
() 17
18 cb=ComboBox(top,label='Type:',editable=Tru
   e)
19 foranimalin('dog','cat','hamster','python'
   ):
20    cb.insert(END,animal)
```

4.4 Related Modules and other GUIs

There are other GUI development systems that can be used with Python.

- Tk-Related Modules
- wxWidgets-Related Modules
- GTK+/GNOME-Related Modules
- Qt/KDE-Related Modules
- Other Open Source GUI Toolkits
- Commercial

4.5 Web Programming – Introduction

Web programming will give you a quick and high-level overview of the kinds of things you can do with Python on the Internet, from Web surfing to creating user feedback forms, from recognizing Uniform Resource Locators to generating dynamic Web page output.

4.6 Web Surfing with Python: Creating Simple Web Clients

Web surfing falls under the same client/server architecture umbrella that we have seen repeatedly. This time, Web clients are browsers, applications that allow users to seek documents on the World Wide Web. On the other side are Web servers, processes that run on an information provider's host computers.

These servers wait for clients and their document requests, process them, and return the requested data. As with most servers in a client/server system, Web servers are designed to run "forever." The Web surfing experience is best illustrated by Figure. Here, a user runs a Web client program such as a browser and makes a connection to a Web server elsewhere on the Internet to obtain information.

A client sends a request out over the Internet to the server, which then responds with the requested data back to the client.

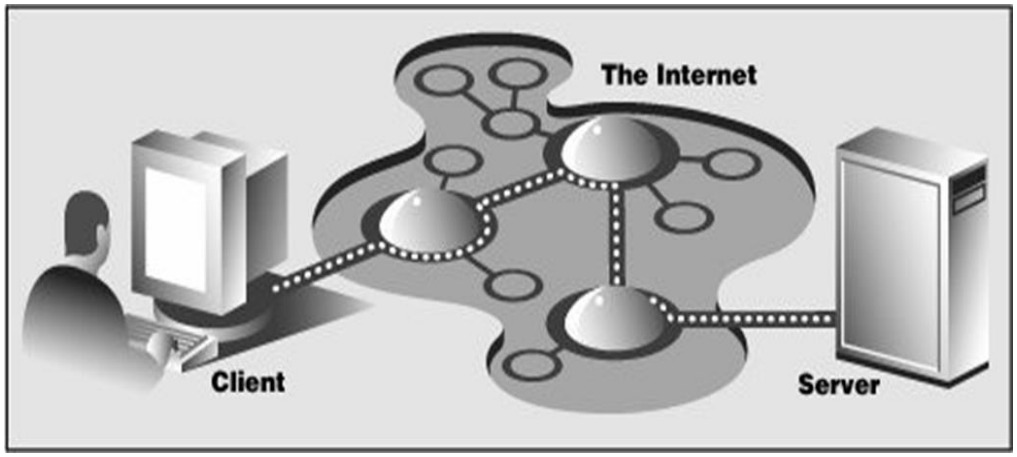

Fig 4.3: Web client and Web server on the Internet.

Clients may issue a variety of requests to Web servers. Such requests may include obtaining a Web page for viewing or submitting a form with data for processing. The request is then serviced by the Web server, and the reply comes back to the client in a special format for display purposes.

The "language" that is spoken by Web clients and servers, the standard protocol used for Web communication, is called HTTP, which stands for HyperText Transfer Protocol. HTTP is written "on top of" the TCP and IP protocol suite, meaning that it relies on TCP and IP to carry out its lower-level communication functionality. Its responsibility is not to route or deliver messagesTCP and IP handle thatbut to respond to client requests (by sending and receiving HTTP messages).

HTTP is known as a "stateless" protocol because it does not keep track of information from one client request to the next, similar to the client/server architecture we have seen so far. The server stays running, but client interactions are singular events structured in such a way that once a client request is serviced, it quits. New requests can always be sent, but they are considered separate service requests. Because of the lack of context per request, you may notice that some URLs have a long set of variablesand values chained as part of the request to provide some sort of state information. Another alternative is the use of "cookies"static data stored on the client side which generally contain state information as well. In later parts of this chapter, we will look at how to use both long URLs and cookies to maintain state information.

4.7 Advanced Web Clients

Web browsers are basic Web clients. They are used primarily for searching and downloading documents from the Web. Advanced clients of the Web are those applications that do more than download single documents from the Internet.

One example of an advanced Web client is a crawler (aka spider, robot). These are programs that explore and download pages from the Internet for different reasons, some of which include:

- Indexing into a large search engine such as Google or Yahoo!
- Offline browsingdownloading documents onto a local hard disk and rearranging hyperlinks to create almost a mirror image for local browsing
- Downloading and storing for historical or archival purposes, or
- Web page caching to save superfluous downloading time on Web site revisits.

The crawler we present below, crawl.py, takes a starting Web address (URL), downloads that page and all other pages whose links appear in succeeding pages, but only those that are in the same domain as the starting page. Without such limitations, you will run out of disk space!

4.8 CGI: Helping Servers Process Client Data

The Web was initially developed to be a global online repository or archive of (mostly educational and research-oriented) documents. Such pieces of information generally come in the form of static text and usually in HTML.

HTML is not as much a language as it is a text formatter, indicating changes in font types, sizes, and styles. The main feature of HTML is in its hypertext capability, text that is in one way or another highlighted to point to another document in a related context to the original. Such a document can be accessed by a mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when and if requested.

As the Internet and Web services evolved, there grew a need to process user input. Online retailers needed to be able to take individual orders, and online banks and search engine portals needed to create accounts for individual users. Thus fill-out forms were invented, and became the only way a Web site can get specific information from users (until Java applets came along). This, in turn, required the HTML now be generated on the fly, for each client submitting user-specific data.

Now, Web servers are only really good at one thing, getting a user request for a file and returning that file (i.e., an HTML file) to the client. They do not have the "brains" to be able to deal with user-specific data such as those which come from fields. Not being their responsibility, Web servers farm out such requests to external applications which create the dynamically generated HTML that is returned to the client.

The entire process begins when the Web server receives a client request (i.e., GET or POST) and calls the appropriate application. It then waits for the resulting HTMLmeanwhile, the client also waits. Once the application has completed, it passes the dynamically generated HTML back to the server, who then (finally) forwards it back to the user. This process of the server receiving a form, contacting an external application, and receiving and returning the newly-generated HTML takes place through what is called the Web server's CGI (Common Gateway Interface). An overview of how CGI works is illustrated in Figure 20-3, which shows you the execution and data flow, step-by-step, from when a user submits aform until the resulting Web page is returned.

CGI represents the interaction between a Web server and the application that is required to process a user's form and generate the dynamic HTML that is eventually returned.

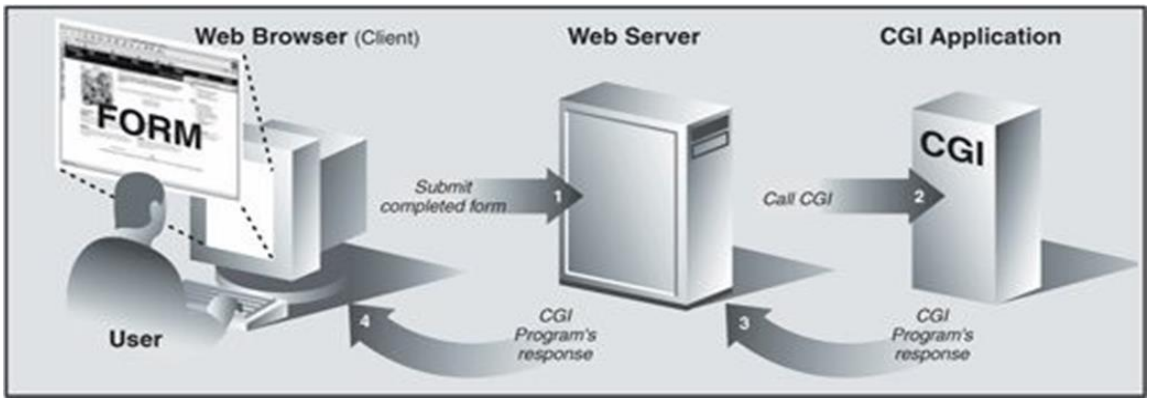

Figure: 4.4 : Overview of how CGI works.

Forms input from the client sent to a Web server may include processing and perhaps some form of storage in a backend database. Just keep in mind that any time there are any user-filled fields and/or a Submit button or image, it most likely involves some sort of CGI activity.

CGI applications that create the HTML are usually written in one of many higher-level programming languages that have the ability to accept user data, process it, and return HTML back to the server. Currently used programming languages include Perl, PHP, C/C++, or Python, to name a few. Before we take a look at CGI, we have to provide the caveat that the typical production Web application is no longer being done in CGI anymore.

Because of its significant limitations and limited ability to allow Web servers to process an abundant number of simultaneous clients, CGI is a dinosaur. Mission-critical Web services rely on compiled languages like C/C++ to scale. A modern-day Web server is typically composed of Apache and integrated components for database access (MySQL or PostgreSQL), Java (Tomcat), PHP, and various modules for Perl, Python, and SSL/security. However, if you are working on small personal Web sites or ones for small organizations and do not need the power and complexity required by mission critical Web services, CGI is the perfect tool for your simple Web sites.

Furthermore, there are a good number of Web application development frameworks out there as well as content management systems, all of which make building CGI a relic of past. However, beneath all the fluff and abstraction, they must still, in the end, follow the same model that CGI originally provided, and that is being able to take user input, execute code based on that input, and provide valid HTML as its final output for the client. Therefore, the exercise in learning CGI is well worth it in terms of understanding the fundamentals in order to develop effective Web services.

4.9 Building CGI Application

In order to play around with CGI development in Python, you need to first install a Web server, configure it for handling Python CGI requests, and then give the Web server access to your CGI scripts. Some of these tasks may require assistance from your system administrator.

If you want a real Web server, you will likely download and install Apache. There are Apache plug-ins or modules for handling Python CGI, but they are not required for our examples. You may wish to install those if you are planning on "going live" to the world with your service. Even this may be overkill.

For learning purposes or for simple Web sites, it may suffice to just use the Web servers that come with Python. In Section 20.8, you will actually learn how to build and configure simple Python-based Webservers. You may read ahead now if you wish to find out more about it at this stage. However, that is not what this section is about.

If you want to just start up the most basic Web server, just execute it directly with Python:

```
$ python -m CGIHTTPServer
```

The -m option is new in 2.4, so if you are using an older version of Python or want to see alternative ways of running it, see section 14.4.3. Anyway, if you eventually get it working.

This will start a Web server on port 8000 on your current machine from the current directory. Then you can just create a Cgi-bin right underneath the directory from which you started the server and put your Python CGI scripts in there. Put some HTML files in that directory and perhaps some .py CGI scripts in Cgi-bin, and you are ready to "surf" directly to this Web site with addresses looking something like these:
http://localhost:8000/friends.htm
http://localhost:8000/cgi-bin/friends2.py

4.10 Advanced CGI

We will now take a look at some of the more advanced aspects of CGI programming. These include the use of cookiescached data saved on the client sidemultiple values for the same CGI field and file upload using multipart form submissions. To save space, we will show you all three of these features with a single application. Let's take a look at multipart submissions first.

Multipart Form Submission and File Uploading

Currently, the CGI specifications only allow two types of form encodings, "application/x-www-form- urlencoded" and "multipart/form-data." Because the former is the default, there is never a need to state the encoding in the FORM tag like this:
<FORM enctype="application/x-www-form-urlencoded" ...>

But for multipart forms, you must explicitly give the encoding as:
<FORM enctype="multipart/form-data" ...>

You can use either type of encoding for form submissions, but at this time, file uploads can only be performed with the multipart encoding. Multipart encoding was invented by Netscape in the early days but has since been adopted by Microsoft (starting with version 4 of Internet Explorer) as well as other browsers.

File uploads are accomplished using the file input type:
<INPUT type=file name=...>

This directive presents an empty text field with a button on the side which allows you to browse your file directory structure for a file to upload. When using multipart, your Web client's form submission to the server will look amazingly like (multipart) e-mail messages with attachments. A separate encoding was needed because it just would not be necessarily wise to "urlencode" a file, especially a binary file. The information still gets to the server, but it is just "packaged" in a different way.

Regardless of whether you use the default encoding or the multipart, the cgi module will process them in the same manner, providing keys and corresponding values in the form submission. You will simply access the data through your FieldStorage instance as before.

Multivalued Fields

In addition to file uploads, we are going to show you how to process fields with multiple values. The most common case is when you have a set of checkboxes allowing a user to select from various choices. Each of the checkboxes is labeled with the same field name, but to differentiate them, each will have a different value associated with a particular checkbox.

As you know, the data from the user are sent to the server in key-value pairs during form submission.

When more than one checkbox is submitted, you will have multiple values associated with the same key. In these cases, rather than being given a single MiniFieldStorage instance for your data, the cgi module will create a list of such instances that you will iterate over to obtain the different values. Not too painful at all.

Cookies

Finally, we will use cookies in our example. If you are not familiar with cookies, they are just bits of data information which a server at a Web site will request to be saved on the client side, e.g., the browser.

Because HTTP is a "stateless" protocol, information that has to be carried from one page to another can be accomplished by using key-value pairs in the request as you have seen in the GET requests and screens earlier in this chapter. Another way of doing it, as we have also seen before, is using hidden form fields, such as the action variable in some of the later friends*.py scripts. These variables and their values are managed by the server because the pages they return to the client must embed these in generated pages.

One alternative to maintaining persistency in state across multiple page views is to save the data on the client side instead. This is where cookies come in. Rather than embedding data to be saved in the returned Web pages, a server will make a request to the client to save a cookie. The cookie is linked to the domain of the originating server (so a server cannot set or override cookies from other Web sites) and has an expiration date (so your browser doesn't become cluttered with cookies).

These two characteristics are tied to a cookie along with the key-value pair representing the data item of interest. There are other attributes of cookies such as a domain subpath or a request that a cookie should only be delivered in a secure environment.

By using cookies, we no longer have to pass the data from page to page to track a user. Although they have been subject to a good amount of controversy over the privacy issue, most Web sites use cookies responsibly. To prepare you for the code, a Web server requests a client store a cookie by sending the "Set-Cookie" header immediately before the requested file.

Once cookies are set on the client side, requests to the server will automatically have those cookies sent to the server using the HTTP_COOKIE environment variable. The cookies are delimited by semicolons and come in "key=value" pairs. All your application needs to do to access the data values is to split the string several times (i.e., using string.split() or manual parsing). The cookies are delimited by semicolons (;), and each key-value pair is separated by equal signs (=).

Like multipart encoding, cookies originated from Netscape, which implemented cookies and wrote up the first specification, which is still valid today. You can access this document at the following Web site:
http://www.netscape.com/newsref/std/cookie_spec.html

Once cookies are standardized and this document finally obsoleted, you will be able to get more current information from Request for Comment documents (RFCs). The most current one for cookies at the time of publication is RFC 2109.

4.11 Web (HTTP) Servers

Until now, we have been discussing the use of Python in creating Web clients and performing tasks to aid Web servers in CGI request processing. We know (and saw earlier in Sections 20.2 and 20.3) that Python can be used to create both simple and complex Web clients. Complexity of CGI requests goes without saying.

However, we have yet to explore the creation of Web servers, and that is the focus of this section. If the Firefox, Mozilla, IE, Opera, Netscape, AOL, Safari, Camino, Epiphany, Galeon, and Lynx browsers are among the most popular Web clients, then what are the most common Web servers? They are Apache, Netscape, IIS, thttpd, Zeus, and Zope. In situations where these servers may be overkill for your desired application, Python can be used to create simple yet useful Web servers.

Creating Web Servers in Python

Since you have decided on building such an application, you will naturally be creating all the custom stuff, but all the base code you will need is already available in the Python Standard Library. To create a Web server, a base server and a "handler" are required.

The base (Web) server is a boilerplate item, a must have. Its role is to perform the necessary HTTP communication between client and server. The base server class is (appropriately) named HTTPServer and is found in the BaseHTTPServer module.

The handler is the piece of software that does the majority of the "Web serving." It processes the client request and returns the appropriate file, whether static or dynamically generated by CGI. The complexity of the handler determines the complexity of your Web server. The Python standard library provides three different handlers.

The most basic, plain, vanilla handler, named BaseHTTPRequestHandler, is found in the BaseHTTPServer module, along with the base Web server. Other than taking a client request, no other handling is implemented at all, so you have to do it all yourself, such as in our myhttpd.py server coming up.

The SimpleHTTPRequestHandler, available in the SimpleHTTP-Server module, builds on BaseHTTPRequestHandler by implementing the standard GET and HEAD requests in a fairly straightforward manner. Still nothing sexy, but it gets the simple jobs done.

Finally, we have the CGIHTTPRequestHandler, available in the CGIHTTPServer module, which takes the SimpleHTTPRequestHandler and adds support for POST requests. It has the ability to call CGI scripts to perform the requested processing and can send the generated HTML back to the client.

UNIT V – DATABASE PROGRAMMING

5.1 Database Programming – Introduction

We are going to cover the Python database API and look at how to access relational databases from Python, either directly through a database interface, or via an ORM, and how you can accomplish the same task but without necessarily having to give explicitly commands in SQL.

Topics such as database principles, concurrency, schema, atomicity, integrity, recovery, proper complex left JOINs, triggers, query optimization, transactions, stored procedures, etc., are all outside the scope of this text, and we will not be discussing these in this chapter other than direct use from a Python application. There are plenty of resources you can refer to for general information. Rather, we will present how to store and retrieve data to/from RDBMSs while playing within a Python framework. You can then decide which is best for your current project or application and be able to study sample code that can get you started instantly. The goal is to get you up to speed as quickly as possible if you need to integrate your Python application with some sort of database system.

We are also breaking out of our mode of covering only the "batteries included" features of the Python standard library. While our original goal was to play only in that arena, it has become clear that being able to work with databases is really a core component of everyday application development in thePython world.

As a software engineer, you can probably only make it so far in your career without having to learn something about databases: how to use one (command-line and/or GUI interfaces), how to pull data out of one using the Structured Query Language (SQL), perhaps how to add or update information in a database, etc. If Python is your programming tool, then a lot of the hard work has already been done for you as you add database access to your Python universe. We first describe what the Python "DB-API" is, then give examples of database interfaces that conform to this standard.

We will give some examples using popular open source relational database management systems (RDBMSs). However, we will not include discussions of open source vs. commercial products, etc. Adapting to those other RDBMS systems should be fairly straightforward. A special mention will be given to Aaron Watters's Gadfly database, a simple RDBMS written completely in Python.

The way to access a database from Python is via an adapter. An adapter is basically a Python module that allows you to interface to a relational database's client library, usually in C. It is recommended that all Python adapters conform to the Python DB-SIG's Application Programmer Interface (API).

The first box is generally a C/C++ program while DB-API compliant adapters let you program applications in Python. ORMs can simplify an application by handling all of the database-specific details.

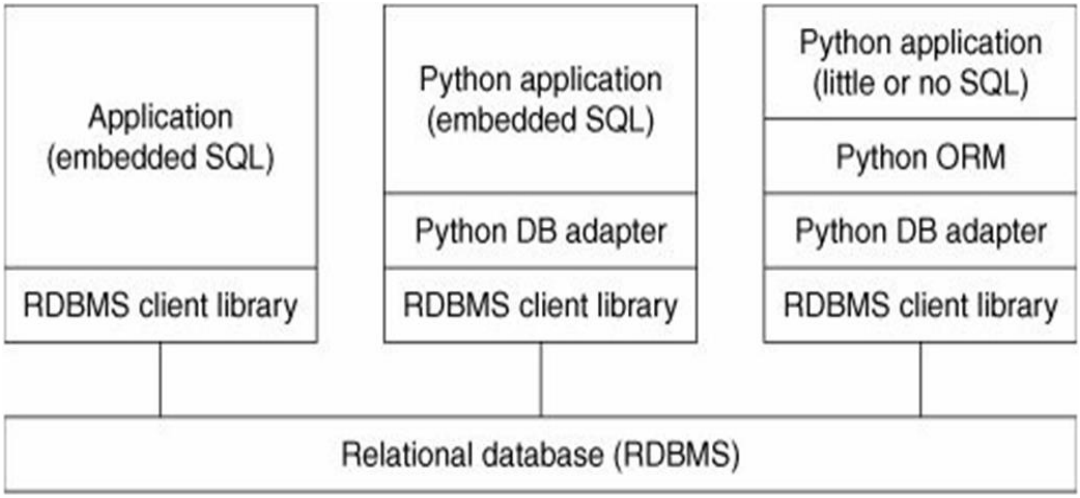

Figure:5.1: Multi-tiered communication between application and database.

5.2 Python Database Application Programmer's Interface (DB-API)

Where can one find the interfaces necessary to talk to a database? Simple. Just go to the database topics section at the main Python Web site. There you will find links to the full and current DB-API (version 2.0), existing database modules, documentation, the special interest group, etc. Since its inception, the DB-API has been moved into PEP 249. (This PEP obsoletes the old DB-API 1.0 specification which is PEP 248.) What is the DB-API?

The API is a specification that states a set of required objects and database access mechanisms to provide consistent access across the various database adapters and underlying database systems. Like most community-based efforts, the API was driven by strong need. In the "old days," we had a scenario of many databases and many people implementing their own database adapters. It was a wheel that was being reinvented over and over again. These databases and adapters were implemented at different times by different people without any consistency of functionality. Unfortunately, this meant that application code using such interfaces also had to be customized to which database module they chose to use, and any changes to that interface also meant updates were needed in the application code.A special interest group (SIG) for Python database connectivity was formed, and eventually, an API was born ... the DB-API version 1.0. The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler, usually only requiring tweaking several lines of code. You will see an example of this later on in this chapter.

Module Attributes

The DB-API specification mandates that the features and attributes listed below must be supplied. A DB- API-compliant module must define the global attributes as shown below.

DB-API Module Attributes

Attribute	**Description**
apilevel	Version of DB-API module is compliant with
threadsafety	Level of thread safety of this module
paramstyle	SQL statement parameter style of this module
Connect()	Connect() function

5.3 Object-Relational Managers (ORMs)

As seen in the previous section, a variety of different database systems are available today, and most of them have Python interfaces to allow you to harness their power. The only drawback to those systems is the need to know SQL. If you are a programmer who feels more comfortable with manipulating Python objects instead of SQL queries, yet still want to use a relational database as your data backend, then you are a great candidate to be a user of ORMs.

Think Objects, Not SQL

Creators of these systems have abstracted away much of the pure SQL layer and implemented objects in Python that you can manipulate to accomplish the same tasks without having to generate the required lines of SQL. Some systems allow for more flexibility if you do have to slip in a few lines of SQL, but for the most part, you can avoid almost all the general SQL required.

Database tables are magically converted to Python classes with columns and features as attributes and methods responsible for database operations. Setting up your application to an ORM is somewhat similar to that of a standard database adapter. Because of the amount of work that ORMs perform on your behalf, some things are actually more complex or require more lines of code than using an adapter directly. Hopefully, the gains you achieve in productivity make up for a little bit of extra work.

Python and ORMs

The most well-known Python ORMs today are SQLAlchemy and SQLObject. We will give you examples of SQLAlchemy and SQLObject because the systems are somewhat disparate due to different philosophies, but once you figure these out, moving on to other ORMs is much simpler.

Some other Python ORMs include PyDO/PyDO2, PDO, Dejavu, PDO, Durus, QLime, and ForgetSQL. Larger Web-based systems can also have their own ORM component, i.e., WebWare MiddleKit and Django's Database API. Note that "well-known" does not mean "best for your application." Although these others were not included in our discussion, that does not mean that they would not be right for your application.

Employee Role Database Example

We will port our user shuffle application ushuffle_db.py to both SQLAlchemy and SQLObject below. MySQL will be the backend database server for both. You will note that we implement these as classes because there is more of an object "feel" to using ORMs as opposed to using raw SQL in a database adapter. Both examples import the set of NAMES and the random name chooser from ushuffle_db.py.
This is to avoid copying-and-pasting the same code everywhere as code reuse is a good thing.

SQLAlchemy

We start with SQLAlchemy because its interface is somewhat closer to SQL than SQLObject's interface. SQLAlchemy abstracts really well to the object world but does give you more flexibility in issuing SQL if you have to. You will find both of these ORMs (Examples 21.2 and 21.3) very similar in terms of setup
and access, as well as being of similar size, and both shorter than ushuffle_db.py (including the sharing of the names list and generator used to randomly iterate through that list).

5.4 Related Modules

Most of the common databases out there along with working Python modules and packages that serve as adapters to those database systems. Note that not all adapters are DB-API- compliant.

Database-Related Modules and Websites

Name	**Online Reference or Description**
Databases	
psycopg	http://initd.org/projects/psycopg1
psycopg2	http://initd.org/software/initd/psycopg/
PyPgSQL	http://pypgsql.sf.net
PyGreSQL	http://pygresql.org
pysqlite	http://initd.org/projects/pysqlite
sqlite3	
sdb	http://dev.mysql.com/downloads/maxdb/7.6.00.html#Python
sapdb	http://sapdb.org/sapdbPython.html
KInterbasDB	http://kinterbasdb.sf.net
pymssql	http://pymssql.sf.net (requires FreeTDS [http://freetds.org])
adodbapi	http://adodbapi.sf.net
sybase	http://object-craft.com.au/projects/sybase
cx_Oracle	http://starship.python.net/crew/atuining/cx_Oracle
DCOracle2	http://zope.org/Members/matt/dco2 (older, for Oracle8 only)
Ingres DBI	http://ingres.com/products/ Prod_Download_Python_DBI.html
ingmod	http://www.informatik.uni-rostock.de/~hme/software/
ORMs	
SQLObject	http://sqlobject.org
SQLAlchemy	http://sqlalchemy.org
PyDO/PyDO2	http://skunkweb.sf.net/pydo.html

Python Programming - Assignment Questions

UNIT-I

1. Define the definition of python programming and explain history of python.
2. How to install python in windows and linux OS.
3. How to work with variables and operators in python.
4. How to read the input from keyboard and explain with a program
5. Explain how decision structures will work in python with example.

UNIT-II

1. Discuss about how to formatting the text output.
2. What are functions to use file input and file ouput.
3. Discuss briefly about defining and calling a void function.
4. Write a program to create, append, and remove lists inpython.
5. Write a program to demonstrate working with tuples inpython.

UNIT-III

1. Summarize the Accessing Characters and Substrings in a String
2. Classify how Testing, Searching, and Manipulating Strings
3. Identify how Finding Items in Lists with the in Operator
4. Discuss short notes on Two-Dimensional Lists
5. Summarize the Problem Solving with Recursion

UNIT-IV

1. Discuss short notes on Classes and Objects
2. Discuss short notes on Classes and Functions
3. Discuss short notes on Classes and Methods
4. Discuss short notes on Working with Instances
5. Discuss short notes on Inheritance and Polymorphism

UNIT-V

1. Summarize the Behavior of terminal based programs and GUI-based programs
2. Write a program by using the tkinter Module
3. Discuss short notes that how Display text with Label Widgets
4. Write a program by using Labels as Output Fields
5. Name the function of Colors and RBG System in python.

Python Programming - Tutorial Questions

UNIT-I

1. Discuss about repetitive structures in python
2. Write a program to demonstrate different number data types inPython.
3. Write a program to perform different Arithmetic Operations on numbers inPython.
4. Write a program to create, concatenate and print a string and accessingsub-string from a givenstring.
5. Write a python script to print the current date in the following format "Sun May 29 02:26:23 IST2017"

UNIT-II

1. Write a program to demonstrate working with dictionaries inpython.
2. Write a python program to find largest of threenumbers.
3. Explain about global variables and global constants.
4. Demonstrate about Value-Returning Functions with Generating Random Numbers
5. How to work with Storing Functions in Modules

UNIT-III

1. Discuss short notes on Dictionaries and Sets
2. Write some Examples of RecursiveAlgorithms
3. Write a Python program to convert temperatures to and from Celsius, Fahrenheit. [Formula : c/5 = f-32/9]
4. Write a Python script that prints prime numbers less than20.
5. Write a python program to find factorial of a number usingRecursion.

UNIT-IV

1. Summarize the features of Procedural and Object-Oriented Programming in python.
2. Write a program that accepts the lengths of three sides of a triangle as inputs. The program output should indicate whether or not the triangle is a right triangle (Recall from the Pythagorean Theorem that in a right triangle, the square of one side equals the sum of the squares of the other twosides).
3. Write a python program to define a module to find Fibonacci Numbers and import the module to anotherprogram.
4. Write a python program to define a module and import a specific function in that module to anotherprogram.
5. Write a script named **copyfile.py**. This script should prompt the user for the names of two text files. The contents of the first file should be input and written to the second file.

UNIT-V

1. Name the Radio Buttons and Check Button functions in python.
2. Write a program that inputs a text file. The program should print all of theunique words in the file in alphabetical order.
3. Write a Python class to convert an integer to a roman numeral.
4. Write a Python class to implement pow(x,n)
5. Write a Python class to reverse a string word byword.

Python Programming - Important Questions

UNIT-I

1. Define the definition of python programming and explain history of python.
2. How to install python in windows and linux OS.
3. How to work with variables and operators in python.
4. How to read the input from keyboard and explain with a program
5. Explain how decision structures will work in python with example.
6. Discuss about repetitive structures in python
7. Write a program to demonstrate different number data types inPython.
8. Write a program to perform different Arithmetic Operations on numbers inPython.
9. Write a program to create, concatenate and print a string and accessingsub-string from a givenstring.
10. Write a python script to print the current date in the following format "Sun May 29 02:26:23 IST2017"

UNIT-II

1. Discuss about how to formatting the text output.
2. What are functions to use file input and file ouput.
3. Discuss briefly about defining and calling a void function.
4. Write a program to create, append, and remove lists inpython.
5. Write a program to demonstrate working with tuples inpython.
6. Write a program to demonstrate working with dictionaries inpython.
7. Write a python program to find largest of threenumbers.
8. Explain about global variables and global constants.
9. Demonstrate about Value-Returning Functions with Generating Random Numbers
10. How to work with Storing Functions in Modules

UNIT-III

1. Summarize the Accessing Characters and Substrings in a String
2. Classify how Testing, Searching, and Manipulating Strings
3. Identify how Finding Items in Lists with the in Operator
4. Discuss short notes on Two-Dimensional Lists
5. Summarize the Problem Solving with Recursion
6. Discuss short notes on Dictionaries and Sets
7. Write some Examples of RecursiveAlgorithms
8. Write a Python program to convert temperatures to and from Celsius, Fahrenheit. [Formula : c/5 = f-32/9]
9. Write a Python script that prints prime numbers less than20.
10. Write a python program to find factorial of a number usingRecursion.

UNIT-IV

1. Discuss short notes on Classes and Objects
2. Discuss short notes on Classes and Functions
3. Discuss short notes on Classes and Methods
4. Discuss short notes on Working with Instances
5. Discuss short notes on Inheritance and Polymorphism
6. Summarize the features of Procedural and Object-Oriented Programming in python.

7. Write a program that accepts the lengths of three sides of a triangle as inputs. The program output should indicate whether or not the triangle is a right triangle (Recall from the Pythagorean Theorem that in a right triangle, the square of one side equals the sum of the squares of the other twosides).
8. Write a python program to define a module to find Fibonacci Numbers and import the module to anotherprogram.
9. Write a python program to define a module and import a specific function in that module to anotherprogram.
10. Write a script named **copyfile.py**. This script should prompt the user for the names of two text files. The contents of the first file should be input and written to the second file.

UNIT-V

1. Summarize the Behavior of terminal based programs and GUI-based programs
2. Write a program by using the tkinter Module
3. Discuss short notes that how Display text with Label Widgets
4. Write a program by using Labels as Output Fields
5. Name the function of Colors and RBG System in python.
6. Name the Radio Buttons and Check Button functions in python.
7. Write a program that inputs a text file. The program should print all of theunique words in the file in alphabetical order.
8. Write a Python class to convert an integer to a romannumeral.
9. Write a Python class to implement pow(x,n)
10. Write a Python class to reverse a string word byword.

PYTHON Programming – Objective Questions

1. Is Python case sensitive when dealing with identifiers?
   a) yes
   b) no
   c) machine dependent
   d) none of the mentioned

Explanation: Case is always significant.

2. What is the maximum possible length of an identifier?
   a) 31 characters
   b) 63 characters
   c) 79 characters
   d) none of the mentioned

Explanation: Identifiers can be of any length.

3. Which of the following is invalid?
   a) _a = 1
   b) a = 1
   c) str = 1
   d) none of the mentioned

Explanation: All the statements will execute successfully but at the cost of reduced readability.

4. Which of the following is an invalid variable?
   a) my_string_1
   b) 1st_string
   c) foo
   d) _

Explanation: Variable names should not start with a number.

5. Why are local variable names beginning with an underscore discouraged?
   a) they are used to indicate a private variables of a class
   b) they confuse the interpreter
   c) they are used to indicate global variables
   d) they slow down execution

Explanation: As Python has no concept of private variables, leading underscores are used to indicate variables that must not be accessed from outside the class.

6. Which of the following is not a keyword?
   a) eval
   b) assert
   c) nonlocal
   d) pass

Explanation: eval can be used as a variable.

7. All keywords in Python are in
   a) lower case
   b) UPPER CASE
   c) Capitalized
   d) None of the mentioned

Explanation: True, False and None are capitalized while the others are in lower case.

8. Which of the following is true for variable names in Python?
   a) unlimited length
   b) all private members must have leading and trailing underscores
   c) underscore and ampersand are the only two special characters allowed
   d) none of the mentioned

Explanation: Variable names can be of any length.

9.In python we do not specify types, it is directly interpreted by the compiler, so consider the following operation to be performed.
>>>x = 13 ? 2
objective is to make sure x has a integer value, select all that apply (python 3.xx)
   a) x = 13 // 2
   b) x = int(13 / 2)
   c) x = 13 % 2
   d) All of the mentioned

Explanation: // is integer operation in python 3.0 and int(..) is a type cast operator.

10. What error occurs when you execute? apple = mango
   a) SyntaxError
   b) NameError
   c) ValueError
   d) TypeError

Explanation: Mango is not defined hence name error.

18. Carefully observe the code and give the answer. def example(a):
a = a + '2'
a = a*2 return a
>>>example("hello")
   a) indentation Error
   b) cannot perform mathematical operation on strings
   c) hello2
   d) hello2hello2

Explanation: Python codes have to be indented properly.

19. What datatype is the object below ?
L = [1, 23, „hello‟, 1].
   a)

Explanation: List datatype can store any values within it.

20. In order to store values in terms of key and value we use what core datatype.
    a) list
    b) tuple
    c) class
    d) dictionary

Explanation: Dictionary stores values in terms of keys and values.

21. Which of the following results in a SyntaxError ?
    a) „"Once upon a time…", she said."
    b) "He said, „Yes!""
    c) '3\'
    d) ""That"s okay""

Explanation: Carefully look at the colons.

22. The following is displayed by a print function call:
    1. tom
    2. dick
    3. harry

Select all of the function calls that result in this output

23. What is the average value of the code that is executed below ?

```
>>>grade1 = 80
>>>grade2 = 90
>>>average = (grade1 + grade2) / 2
```

    a) 85
    b) 85.1
    c) 95
    d) 95.1

Explanation: Cause a decimal value to appear as output.

24. Select all options that print hello-how-are-you

a) print(„hello", „how", „are", „you")
b) print(„hello", „how", „are", „you" + „-„ * 4)
c) print('hello-' + 'how-are-you')
d) print(„hello" + „-„ + „how" + „-„ + „are" + „you")

Explanation: Execute in the shell.

25. What is the return value of trunc() ?

a) int
b) bool
c) float
d) None

Explanation: Executle help(math.trunc) to get details.

26. Which is the correct operator for power(x^y)?

a) X^y
b) X**y
c) X^^y
d) None of the mentioned

Explanation: In python, power operator is x**y i.e. 2**3=8.

27. Which one of these is floor division?
a) /
b) //
c) %
d) None of the mentioned
Explanation: When both of the operands are integer then python chops out the fraction part and gives you the round off value, to get the accurate answer use floor division. This is floor division. For ex, 5/2 = 2.5 but both of the operands are integer so answer of this expression in python is 2.To get the 2.5 answer, use floor division.

28. What is the order of precedence in python?
i) Parentheses
ii) Exponential
iii) Division
iv) Multiplication
v) Addition
vi) Subtraction
a) i,ii,iii,iv,v,vi
b) ii,i,iii,iv,v,vi
c) ii,i,iv,iii,v,vi
d) i,ii,iii,iv,vi,v
Explanation: For order of precedence, just remember this PEDMAS (similar to BODMAS)

29. What is answer of this expression, 22 % 3 is?
a) 7
b) 1
c) 0
d) 5
Explanation: Modulus operator gives remainder. So, 22%3 gives the remainder, that is, 1.

30. Mathematical operations can be performed on a string. State whether true or false.
a) True
b) False
Explanation: You can‟t perform mathematical operation on string even if the string is in the form: „1234…‟.

31. Operators with the same precedence are evaluated in which manner?
a) Left to Right
b) Right to Left
c) Cant say
d) None of the mentioned
Explanation: None.

32. What is the output of this expression, 3*1**3?
a) 27
b) 9
c) 3
d) 1

33. Which one of the following have the same precedence?
a) Addition and Subtraction
b) Multiplication and Division
c) Both a and b
d) None of the mentioned
Explanation: None.

34. The expression Int(x) implies that the variable x is converted to integer. State whether true or false.
a) True
b) False
Explanation: None.

35. Which one of the following have the highest precedence in the expression?
a) Exponential
b) Addition
c) Multiplication
d) Parentheses
Explanation: Just remember: PEDMAS, that is, Parenthesis, Exponentiation, Division, Multiplication, Addition, Subtraction. Note that the precedence order of Division and Multiplication is the same. Likewise, the order of Addition and Subtraction is also the same.

36. What is the output of print 0.1 + 0.2 == 0.3?
a) True
b) False
c) Machine dependent
d) Error
Explanation: Neither of 0.1, 0.2 and 0.3 can be represented accurately in binary. The round off errors from 0.1 and 0.2 accumulate and hence there is a difference of 5.5511e-17 between (0.1 + 0.2) and 0.3.
37. Which of the following is not a complex number?
a) k = 2 + 3j
b) k = complex(2, 3)
c) k = 2 + 3l
d) k = 2 + 3J
Explanation: l (or L) stands for long.

38. What does ~4 evaluate to?
a) -5
b) -4
c) -3
d) +3
Explanation: ~x is equivalent to -(x+1).

39. What does ~~~~~~5 evaluate to?
a) +5
b) -11
c) +11
d) -5
Explanation: ~x is equivalent to -(x+1).

40. Which of the following is incorrect?
a) x = 0b101
b) x = 0x4f5
c) x = 19023
d) x = 03964
Explanation: Numbers starting with a 0 are octal numbers but 9 isn‟t allowed in octal numbers.

41. What is the result of cmp(3, 1)?
a) 1
b) 0
c) True
d) False
Explanation: cmp(x, y) returns 1 if x > y, 0 if x == y and -1 if x < y.

42. Which of the following is incorrect?
a) float(„inf‟)
b) float(„nan‟)
c) float(“56“+“78“)
d) float('12+34′)
Explanation: „+‟ cannot be converted to a float.

43. What is the result of round(0.5) – round(-0.5)?
a) 1.0
b) 2.0
c) 0.0
d) None of the mentioned
Explanation: Python rounds off numbers away from 0 when the number to be rounded off is exactly halfway through. round(0.5) is 1 and round(-0.5) is -1.

44. What does 3 ^ 4 evaluate to?
a) 81
b) 12
c) 0.75
d) 7
Explanation: ^ is the Binary XOR operator

45. Which module in Python supports regular expressions?
a) re
b) regex
c) pyregex
d) none of the mentioned
Explanation: re is a part of the standard library and can be imported using: import re.
46. What does the function re.match do?
a) matches a pattern at the start of the string
b) matches a pattern at any position in the string
c) such a function does not exist
d) none of the mentioned
Explanation: It will look for the pattern at the beginning and return None if it isn‟t found.

47. What does the function re.search do?
a) matches a pattern at the start of the string
b) matches a pattern at any position in the string
c) such a function does not exist
d) none of the mentioned
Explanation: It will look for the pattern at any position in the string.

48. What is the output of the following?
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence) print(matched.groups())
a) (‘we’, ‘are’, ‘humans’)
b) (we, are, humans)
c) („we‟, „humans‟)
d) „we are humans‟
Explanation: This function returns all the subgroups that have been matched.

Previous Question Papers- PYTHON Programming

Code No: R1621054 **R16** SET - 1

II B. Tech I Semester Regular Examinations, October/November - 2017
PYTHON PROGRAMMING
(Com to CSE & IT)

Time: 3 hours Max. Marks: 70

Note: 1. Question Paper consists of two parts (**Part-A** and **Part-B**)
2. Answer **ALL** the question in **Part-A**
3. Answer any **FOUR** Questions from **Part-B**

~~~~~~

PART –A

1.	a)	Explain input function.	(2M)
	b)	Give an example of lstrip( ) method.	(2M)
	c)	How to access values in a dictionary?	(2M)
	d)	What is default argument?	(2M)
	e)	What are basic overloading methods?	(3M)
	f)	Explain importing turtle graphics.	(3M)

PART -B

2.	a)	What are IDLE usability features?	(7M)
	b)	Explain about keywords used in Python.	(7M)
3.	a)	What are 4 built-in numeric data types in Python? Explain.	(7M)
	b)	Describe Python jump statements with examples.	(7M)
4.	a)	Explain in detail about dictionaries in Python.	(7M)
	b)	Discuss about tuples in Pyhton.	(7M)
5.	a)	Describe anonymous functions examples.	(7M)
	b)	Why to use modules? How to structure a program?	(7M)
6.	a)	Explain creating classes in Python with examples.	(7M)
	b)	Define error and exception. Distinguish between these two features.	(7M)
7.	a)	Why testing is required? Explain in detail.	(7M)
	b)	Explain the following: i) Calendar module ii) Synchronizing threads	(7M)

1 of 1
~~~~~~