Appendix A. A Selective and Impressionistic Short Review of Python

A reader who is coming to Python for the first time would be well served reading Guido van Rossum's *Python Tutorial*, which can be downloaded from <http://python.org/>, or picking up one of the several excellent books devoted to teaching Python to novices. As indicated in the Preface, the audience of this book is a bit different.

The above said, some readers of this book might use Python only infrequently, or not have used Python for a while, or may be sufficiently versed in numerous other programming languages, that a quick review on Python constructs suffices for understanding. This appendix will briefly mention each major element of the Python language itself, but will not address any libraries (even standard and ubiquitous ones that may be discussed in the main chapters). Not all fine points of syntax and semantics will be covered here, either. This review, however, should suffice for a reader to understand all the examples in this book.

Even readers who are familiar with Python might enjoy skimming this review. The focus and spin of this summary are a bit different from most introductions. I believe that the way I categorize and explain a number of language features can provide a moderately novel—but equally accurate—perspective on the Python language. Ideally, a Python programmer will come away from this review with a few new insights on the familiar constructs she uses every day. This appendix does not shy away from using some abstract terms from computer science—if a particular term is not familiar to you, you will not lose much by skipping over the sentence it occurs in; some of these terms are explained briefly in the Glossary.

A.1 What Kind of Language Is Python?

Python is a byte-code compiled programming language that supports multiple programming paradigms. Python is sometimes called an interpreted and/or scripting language because no separate compilation step is required to run a Python program; in more precise terms, Python uses a virtual machine (much like Java or Smalltalk) to run machine-abstracted instructions. In most situations a byte-code compiled version of an application is cached to speed future runs, but wherever necessary compilation is performed "behind the scenes."

In the broadest terms, Python is an imperative programming language, rather than a declarative (functional or logical) one. Python is dynamically and strongly typed, with very late binding compared to most languages. In addition, Python is an object-oriented language with strong introspective facilities, and one that generally relies on conventions rather than enforcement mechanisms to control access and visibility of names. Despite its object-oriented core, much of the syntax of Python is designed to allow a convenient procedural style that masks the underlying OOP mechanisms. Although Python allows basic functional programming (FP) techniques, side effects are the norm, evaluation is always strict, and no compiler optimization is performed for tail recursion (nor on almost any other construct).

Python has a small set of reserved words, delimits blocks and structure based on indentation only, has a fairly rich collection of built-in data structures, and is generally both terse and readable compared to other programming languages. Much of the strength of Python lies in its standard library and in a flexible system of importable modules and packages.

A.2 Namespaces and Bindings

The central concept in Python programming is that of a namespace. Each context (i.e., scope) in a Python program has available to it a hierarchically organized collection of namespaces; each namespace contains a set of names, and each name is bound to an object. In older versions of Python, namespaces were arranged according to the "three-scope rule" (builtin/global/local), but Python version 2.1 and later add lexically nested scoping. In most cases you do not need to worry about this subtlety, and scoping works the way you would expect (the special cases that prompted the addition of lexical scoping are mostly ones with nested functions and/or classes).

There are quite a few ways of binding a name to an object within the current namespace/scope and/or within some other scope. These various ways are listed below.

A.2.1 Assignment and Dereferencing

A Python statement like `x=37` or `y="foo"` does a few things. If an object—e.g. `37` or `"foo"`—does not exist, Python creates one. If such an object *does* exist, Python locates it. Next, the name `x` or `y` is added to the current namespace, if it does not exist already, and that name is bound to the corresponding object. If a name already exists in the current namespace, it is re-bound. Multiple names, perhaps in multiple scopes/namespaces, can be bound to the same object.

A simple assignment statement binds a name into the current namespace, unless that name has been declared as global. A name declared as global is bound to the global (module-level) namespace instead. A qualified name used on the left of an assignment statement binds a name into a specified namespace—either to the attributes of an object, or to the namespace of a module/package; for example:

```
>>> x = "foo"             # bind 'x' in global namespace
>>> def myfunc():         # bind 'myfunc' in global namespace
...     global x, y       # specify namespace for 'x', 'y'
...     x = 1             # rebind global 'x' to 1 object
...     y = 2             # create global name 'y' and 2 object
...     z = 3             # create local name 'z' and 3 object
...
>>> import package.module  # bind name 'package.module'
>>> package.module.w = 4   # bind 'w' in namespace package.module
>>> from mymod import obj  # bind object 'obj' to global namespace
>>> obj.attr = 5          # bind name 'attr' to object 'obj'
```

Whenever a (possibly qualified) name occurs on the right side of an assignment, or on a line by itself, the name is dereferenced to the object itself. If a name has not been bound inside some accessible scope, it cannot be dereferenced; attempting to do so raises a `NameError` exception. If the name is followed by left and right parentheses (possibly with comma-separated expressions between them), the object is invoked/called after it is dereferenced. Exactly what happens upon invocation can be controlled and overridden for Python objects; but in general, invoking a function or method runs some code, and invoking a class creates an instance. For example:

```
>>> pkg.subpkg.func()   # invoke a function from a namespace
>>> x = y               # deref 'y' and bind same object to 'x'
```

A.2.2 Function and Class Definitions

Declaring a function or a class is simply the preferred way of describing an object and binding it to a name. But the `def` and `class` declarations are "deep down" just types of assignments. In the case of functions, the *lambda* operator can also be used on the right of an assignment to bind an "anonymous" function to a name. There is no equally direct technique for classes, but their declaration is still

similar in effect:

```
>>> add1 = lambda x,y: x+y # bind 'add1' to function in global ns
>>> def add2(x, y):        # bind 'add2' to function in global ns
...     return x+y
...
>>> class Klass:           # bind 'Klass' to class object
...     def meth1(self):    # bind 'meth1' to method in 'Klass' ns
...         return 'Myself'
```

A.2.3 `import` Statements

Importing, or importing *from,* a module or a package adds or modifies bindings in the current namespace. The `import` statement has two forms, each with a bit different effect.

Statements of the forms

```
>>> import modname
>>> import pkg.subpkg.modname
>>> import pkg.modname as othername
```

add a new module object to the current namespace. These module objects themselves define namespaces that you can bind values in or utilize objects within.

Statements of the forms

```
>>> from modname import foo
>>> from pkg.subpkg.modname import foo as bar
```

instead add the names `foo` or `bar` to the current namespace. In any of these forms of `import`, any statements in the imported module are executed—the difference between the forms is simply the effect upon namespaces.

There is one more special form of the `import` statement; for example:

```
>>> from modname import *
```

The asterisk in this form is not a generalized glob or regular expression pattern, it is a special syntactic form. "Import star" imports every name in a module namespace into the current namespace (except those named with a leading underscore, which can still be explicitly imported if needed). Use of this form is somewhat discouraged because it risks adding names to the current namespace that you do not explicitly request and that may rebind existing names.

A.2.4 `for` Statements

Although `for` is a looping construct, the way it works is by binding successive elements of an iterable object to a name (in the current namespace). The following constructs are (almost) equivalent:

```
>>> for x in somelist:  # repeated binding with 'for'
...     print x
...
>>> ndx = 0             # rebinds 'ndx' if it was defined
>>> while 1:            # repeated binding in 'while'
...     x = somelist[ndx]
...     print x
...     ndx = ndx+1
```

```
...     if ndx >= len(somelist):
...         del ndx
...         break
```

A.2.5 except Statements

The except statement can optionally bind a name to an exception argument:

```
>>> try:
...     raise "ThisError", "some message"
... except "ThisError", x:    # Bind 'x' to exception argument
...     print x
...
some message
```

A.3 Datatypes

Python has a rich collection of basic datatypes. All of Python's collection types allow you to hold heterogeneous elements inside them, including other collection types (with minor limitations). It is straightforward, therefore, to build complex data structures in Python.

Unlike many languages, Python datatypes come in two varieties: mutable and immutable. All of the atomic datatypes are immutable, as is the collection type tuple. The collections list and dict are mutable, as are class instances. The mutability of a datatype is simply a question of whether objects of that type can be changed "in place"—an immutable object can only be created and destroyed, but never altered during its existence. One upshot of this distinction is that immutable objects may act as dictionary keys, but mutable objects may not. Another upshot is that when you want a data structure—especially a large one—that will be modified frequently during program operation, you should choose a mutable datatype (usually a list).

Most of the time, if you want to convert values between different Python datatypes, an explicit conversion/encoding call is required, but numeric types contain promotion rules to allow numeric expressions over a mixture of types. The built-in datatypes are listed below with discussions of each. The built-in function *type()* can be used to check the datatype of an object.

A.3.1 Simple Types

bool

Python 2.3+ supports a Boolean datatype with the possible values True and False. In earlier versions of Python, these values are typically called 1 and 0; even in Python 2.3+, the Boolean values behave like numbers in numeric contexts. Some earlier micro-releases of Python (e.g., 2.2.1) include the *names* True and False, but not the Boolean datatype.

int

A signed integer in the range indicated by the register size of the interpreter's CPU/OS platform. For most current platforms, integers range from (2**31)-1 to negative (2**31)-1. You can find the size on your platform by examining *sys.maxint*. Integers are the bottom numeric type in terms of promotions; nothing gets promoted *to* an integer, but integers are sometimes promoted to other numeric types. A float, long, or string may be explicitly converted to an int using the *int()* function.

SEE ALSO: int *18;*

long

An (almost) unlimited size integral number. A long literal is indicated by an integer followed by an l or L (e.g., 34L, 98765432101). In

Python 2.2+, operations on ints that overflow *sys.maxint* are automatically promoted to longs. An int, float, or string may be explicitly converted to a long using the *long()* function.

float

An IEEE754 floating point number. A literal floating point number is distinguished from an int or long by containing a decimal point and/or exponent notation (e.g., 1.0, 1e3, 37., .453e-12). A numeric expression that involves both int/long types and float types promotes all component types to floats before performing the computation. An int, long, or string may be explicitly converted to a float using the *float()* function.

SEE ALSO: float *19;*

complex

An object containing two floats, representing real and imaginary components of a number. A numeric expression that involves both int/long/float types and complex types promotes all component types to complex before performing the computation. There is no way to spell a literal complex in Python, but an addition such as 1.1+2j is the usual way of computing a complex value. Aj or J following a float or int literal indicates an imaginary number. An int, long, or string may be explicitly converted to a complex using the *complex()* function. If two float/int arguments are passed to *complex()*, the second is the imaginary component of the constructed number (e.g., complex(1.1,2)).

string

An immutable sequence of 8-bit character values. Unlike in many programming languages, there is no "character" type in Python, merely strings that happen to have length one. String objects have a variety of methods to modify strings, but such methods always return a new string object rather than modify the initial object itself. The built-in *chr()* function will return a length-one string whose ordinal value is the passed integer. The *str()* function will return a string representation of a passed in object. For example:

```
>>> ord('a')
97
>>> chr(97)
'a'
>>> str(97)
'97'
```

SEE ALSO: string *129;*

unicode

An immutable sequence of Unicode characters. There is no datatype for a single Unicode character, but Unicode strings of length-one contain a single character. Unicode strings contain a similar collection of methods to string objects, and like the latter, Unicode methods return new Unicode objects rather than modify the initial object. See Chapter 2 and Appendix C for additional discussion, of Unicode.

A.3.2 String Interpolation

Literal strings and Unicode strings may contain embedded format codes. When a string contains format codes, values may be *interpolated* into the string using the% operator and a tuple or dictionary giving the values to substitute in.

Strings that contain format codes may follow either of two patterns. The simpler pattern uses format codes with the syntax %[flags][len[.precision]]<type>. Interpolating a string with format codes on this pattern requires% combination with a tuple of matching length and content datatypes. If only one value is being interpolated, you may give the bare item rather than a tuple of length one. For example:

```
>>> "float %3.1f, int %+d, hex %06x" % (1.234, 1234, 1234)
'float 1.2, int +1234, hex 0004d2'
>>> '%e' % 1234
'1.234000e+03'
>>> '%e' % (1234,)
'1.234000e+03'
```

The (slightly) more complex pattern for format codes embeds a name within the format code, which is then used as a string key to an interpolation dictionary. The syntax of this pattern is %(key)[flags][len[.precision]]<type>. Interpolating a string with this style of format codes requires % combination with a dictionary that contains all the named keys, and whose corresponding values contain acceptable datatypes. For example:

```
>>> dct = {'ratio':1.234, 'count':1234, 'offset':1234}
>>> "float %(ratio)3.1f, int %(count)+d, hex %(offset)06x" % dct
'float 1.2, int +1234, hex 0004d2'
```

You *may not* mix tuple interpolation and dictionary interpolation within the same string.

I mentioned that datatypes must match format codes. Different format codes accept a different range of datatypes, but the rules are almost always what you would expect. Generally, numeric data will be promoted or demoted as necessary, but strings and complex types cannot be used for numbers.

One useful style of using dictionary interpolation is against the global and/or local namespace dictionary. Regular bound names defined in scope can be interpolated into strings.

```
>>> s = "float %(ratio)3.1f, int %(count)+d, hex %(offset)06x"
>>> ratio = 1.234
>>> count = 1234
>>> offset = 1234
>>> s % globals()
'float 1.2, int +1234, hex 0004d2'
```

If you want to look for names across scope, you can create an ad hoc dictionary with both local and global names:

```
>>> vardct = {}
>>> vardct.update(globals())
>>> vardct.update(locals())
>>> interpolated = somestring % vardct
```

The flags for format codes consist of the following:

```
0 Pad to length with leading zeros
- Align the value to the left within its length
- (space) Pad to length with leading spaces
+ Explicitly indicate the sign of positive values
```

When a length is included, it specifies the *minimum* length of the interpolated formatting. Numbers that will not fit within a length simply occupy more bytes than specified. When a precision is included, the length of those digits to the right of the decimal are included in the total length:

```
>>> '[%f]' % 1.234
'[1.234000]'
>>> '[%5f]' % 1.234
'[1.234000]'
>>> '[%.1f]' % 1.234
'[1.2]'
>>> '[%5.1f]' % 1.234
'[  1.2]'
>>> '[%05.1f]' % 1.234
'[001.2]'
```

The formatting types consist of the following:

```
d Signed integer decimal
i Signed integer decimal
o Unsigned octal
u Unsigned decimal
x Lowercase unsigned hexadecimal
X Uppercase unsigned hexadecimal
e Lowercase exponential format floating point
E Uppercase exponential format floating point
f Floating point decimal format
g Floating point: exponential format if -4 < exp < precision
G Uppercase version of 'g'
c Single character: integer for chr(i) or length-one string
r Converts any Python object using repr()
s Converts any Python object using str()
% The '%' character, e.g.: '%%%d' % (1) --> '%1'
```

One more special format code style allows the use of a * in place of a length. In this case, the interpolated tuple must contain an extra element for the formatted length of each format code, preceding the value to format. For example:

```
>>> "%0*d # %0*.2f" % (4, 123, 4, 1.23)
'0123 # 1.23'
>>> "%0*d # %0*.2f" % (6, 123, 6, 1.23)
'000123 # 001.23'
```

A.3.3 Printing

The least-sophisticated form of textual output in Python is writing to open files. In particular, the STDOUT and STDERR streams can be accessed using the pseudo-files *sys.stdout* and *sys.stderr*. Writing to these is just like writing to any other file; for example:

```
>>> import sys
>>> try:
...     # some fragile action
...     sys.stdout.write('result of action\n')
... except:
...     sys.stderr.write('could not complete action\n')
...
result of action
```

You cannot seek within STDOUT or STDERR—generally you should consider these as pure sequential outputs.

Writing to STDOUT and STDERR is fairly inflexible, and most of the time the `print` statement accomplishes the same purpose more flexibly. In particular, methods like *sys.stdout.write()* only accept a single string as an argument, while `print` can handle any number of arguments of any type. Each argument is coerced to a string using the equivalent of `repr(obj)`. For example:

```
>>> print "Pi: %.3f" % 3.1415, 27+11, {3:4,1:2}, (1,2,3)
Pi: 3.142 38 {1: 2, 3: 4} (1, 2, 3)
```

Each argument to the print statment is evaluated before it is printed, just as when an argument is passed to a function. As a consequence, the canonical representation of an object is printed, rather than the exact form passed as an argument. In my example, the dictionary prints in a different order than it was defined in, and the spacing of the list and dictionary is slightly different. String interpolation is also peformed and is a very common means of defining an output format precisely.

There are a few things to watch for with the print statement. A space is printed between each argument to the statement. If you want to print several objects without a separating space, you will need to use string concatenation or string interpolation to get the right result. For example:

```
>>> numerator, denominator = 3, 7
>>> print repr(numerator)+"/"+repr(denominator)
3/7
>>> print "%d/%d" % (numerator, denominator)
3/7
```

By default, a print statement adds a linefeed to the end of its output. You may eliminate the linefeed by adding a trailing comma to the statement, but you still wind up with a space added to the end:

```
>>> letlist = ('a','B','Z','r','w')
>>> for c in letlist: print c,   # inserts spaces
...
a B Z r w
```

Assuming these spaces are unwanted, you must either use *sys.stdout.write()* or otherwise calculate the space-free string you want:

```
>>> for c in letlist+('\n',): # no spaces
...     sys.stdout.write(c)
...
aBZrw
>>> print ''.join(letlist)
aBZrw
```

There is a special form of the print statement that redirects its output somewhere other than STDOUT. Theprint statement itself can be followed by two greater-than signs, then a writable file-like object, then a comma, then the remainder of the (printed) arguments. For example:

```
>>> print >> open('test','w'), "Pi: %.3f" % 3.1415, 27+11
>>> open('test').read()
'Pi: 3.142 38\n'
```

Some Python programmers (including your author) consider this special form overly "noisy," but it *is* occassionally useful for quick configuration of output destinations.

If you want a function that would do the same thing as a print statement, the following one does so, but without any facility to eliminate the trailing linefeed or redirect output:

```
def print_func(*args):
    import sys
    sys.stdout.write(' '.join(map(repr,args))+'\n')
```

Readers could enhance this to add the missing capabilities, but using print as a statement is the clearest approach, generally.

SEE ALSO: sys.stderr *50;* sys.stdout *51;*

A.3.4 Container Types

tuple

An immutable sequence of (heterogeneous) objects. Being immutable, the membership and length of a tuple cannot be modified after creation. However, tuple elements and subsequences can be accessed by subscripting and slicing, and new tuples can be constructed from such elements and slices. Tuples are similar to "records" in some other programming languages.

The constructor syntax for a tuple is commas between listed items; in many contexts, parentheses around a constructed list are required to disambiguate a tuple for other constructs such as function arguments, but it is the commas not the parentheses that construct a tuple. Some examples:

```
>>> tup = 'spam','eggs','bacon','sausage'
>>> newtup = tup[1:3] + (1,2,3) + (tup[3],)
>>> newtup
('eggs', 'bacon', 1, 2, 3, 'sausage')
```

The function *tuple()* may also be used to construct a tuple from another sequence type (either a list or custom sequence type).

SEE ALSO: tuple *28;*

list

A mutable sequence of objects. Like a tuple, list elements can be accessed by subscripting and slicing; unlike a tuple, list methods and index and slice assignments can modify the length and membership of a list object.

The constructor syntax for a list is surrounding square braces. An empty list may be constructed with no objects between the braces; a length-one list can contain simply an object name; longer lists separate each element object with commas. Indexing and slices, of course, also use square braces, but the syntactic contexts are different in the Python grammar (and common sense usually points out the difference). Some examples:

```
>>> lst = ['spam', (1,2,3), 'eggs', 3.1415]
>>> lst[:2]
['spam', (1, 2, 3)]
```

The function *list()* may also be used to construct a list from another sequence type (either a tuple or custom sequence type).

SEE ALSO: list *28;*

dict

A mutable mapping between immutable keys and object values. At most one entry in a dict exists for a given key; adding the same key to a dictionary a second time overrides the previous entry (much as with binding a name in a namespace). Dicts are unordered, and entries are accessed either by key as index; by creating lists of contained objects using the methods .keys(), .values(), and .items(); or—in recent Python versions—with the .popitem() method. All the dict methods generate contained objects in an unspecified order.

The constructor syntax for a dict is surrounding curly brackets. An empty dict may be constructed with no objects between the brackets. Each key/value pair entered into a dict is separated by a colon, and successive pairs are separated by commas. For example:

```
>>> dct = {1:2, 3.14:(1+2j), 'spam':'eggs'}
>>> dct['spam']
'eggs'
```

```
>>> dct['a'] = 'b'    # add item to dict
>>> dct.items()
[('a', 'b'), (1, 2), ('spam', 'eggs'), (3.14, (1+2j))]
>>> dct.popitem()
('a', 'b')
>>> dct
{1: 2, 'spam': 'eggs', 3.14: (1+2j)}
```

In Python 2.2+, the function *dict()* may also be used to construct a dict from a sequence of pairs or from a custom mapping type. For example:

```
>>> d1 = dict([('a','b'), (1,2), ('spam','eggs')])
>>> d1
{'a': 'b', 1: 2, 'spam': 'eggs'}
>>> d2 = dict(zip([1,2,3],['a','b','c']))
>>> d2
{1: 'a', 2: 'b', 3: 'c'}
```

SEE ALSO: dict*24;*

sets.Set

Python 2.3+ includes a standard module that implements a set datatype. For earlier Python versions, a number of developers have created third-party implementations of sets. If you have at least Python 2.2, you can download and use the *sets* module from <http://tinyurl.com/2d31> (or browse the Python CVS)—you will need to add the definition`True,False=1, 0` to your local version, though.

A set is an unordered collection of hashable objects. Unlike a list, no object can occur in a set more than once; a set resembles a dict that has only keys but no values. Sets utilize bitwise and Boolean syntax to perform basic set-theoretic operations; a subset test does not have a special syntactic form, instead using the `.issubset()` and `.issuperset()` methods. You may also loop through set members in an unspecified order. Some examples illustrate the type:

```
>>> from sets import Set
>>> x = Set([1,2,3])
>>> y = Set((3,4,4,6,6,2)) # init with any seq
>>> print x, '//', y       # make sure dups removed
Set([1, 2, 3]) // Set([2, 3, 4, 6])
>>> print x | y            # union of sets
Set([1, 2, 3, 4, 6])
>>> print x & y            # intersection of sets
Set([2, 3])
>>> print y-x              # difference of sets
Set([4, 6])
>>> print x ^ y            # symmetric difference
Set([1, 4, 6])
```

You can also check membership and iterate over set members:

```
>>> 4 in y                # membership check
1
>>> x.issubset(y)         # subset check
0
>>> for i in y:
...     print i+10,
...
12 13 14 16
>>> from operator import add
>>> plus_ten = Set(map(add, y, [10]*len(y)))
>>> plus_ten
```

```
Set([16, 12, 13, 14])
```

sets.Set also supports in-place modification of sets; *sets.ImmutableSet*, naturally, does not allow modification.

```
>>> x = Set([1,2,3])
>>> x |= Set([4,5,6])
>>> x
Set([1, 2, 3, 4, 5, 6])
>>> x &= Set([4,5,6])
>>> x
Set([4, 5, 6])
>>> x ^= Set ([4, 5])
>>> x
Set([6])
```

A.3.5 Compound Types

class instance

A class instance defines a namespace, but this namespace's main purpose is usually to act as a data container (but a container that also knows how to perform actions; i.e., has methods). A class instance (or any namespace) acts very much like a dict in terms of creating a mapping between names and values. Attributes of a class instance may be set or modified using standard qualified names and may also be set within class methods by qualifying with the namespace of the first (implicit) method argument, conventionally called self. For example:

```
>>> class Klass:
...     def setfoo(self, val):
...         self.foo = val
...
>>> obj = Klass()
>>> obj.bar = 'BAR'
>>> obj.setfoo(['this','that','other'])
>>> obj.bar, obj.foo
('BAR', ['this', 'that', 'other'])
>>> obj.__dict__
{'foo': ['this', 'that', 'other'], 'bar': 'BAR'}
```

Instance attributes often dereference to other class instances, thereby allowing hierarchically organized namespace quantification to indicate a data structure. Moreover, a number of "magic" methods named with leading and trailing double-underscores provide optional syntactic conveniences for working with instance data. The most common of these magic methods is .__init__(), which initializes an instance (often utilizing arguments). For example:

```
>>> class Klass2:
...     def __init__(self, *args, **kw):
...         self.listargs = args
...         for key, val in kw.items():
...             setattr(self, key, val)
...
>>> obj = Klass2(1, 2, 3, foo='F00', bar=Klass2(baz='BAZ'))
>>> obj.bar.blam = 'BLAM'
>>> obj.listargs, obj.foo, obj.bar.baz, obj.bar.blam
((1, 2, 3), 'F00', 'BAZ', 'BLAM')
```

There are quite a few additional "magic" methods that Python classes may define. Many of these methods let class instances behave more like basic datatypes (while still maintaining special class behaviors). For example, the .__str__() and .__repr__() methods control the string representation of an instance; the .__getitem__() and .__setitem__() methods allow indexed access to instance data (either dict-like named indices, or list-like numbered indices); methods like .__add__(), .__mul__(), .__pow__(), and .__abs__() allow instances to behave in number-like ways. The *Python Reference Manual* discusses magic methods in detail.

In Python 2.2 and above, you can also let instances behave more like basic datatypes by inheriting classes from these built-in types. For example, suppose you need a datatype whose "shape" contains both a mutable sequence of elements and a .foo attribute. Two ways to define this datatype are:

```
>>> class FooList(list):        # works only in Python 2.2+
...     def __init__(self, lst=[], foo=None):
...         list.__init__(self, lst)
...         self.foo = foo
...
>>> foolist = FooList([1,2,3], 'F00')
>>> foolist[1], foolist.foo
(2, 'F00')
>>> class oldFooList:           # works in older Pythons
...     def __init__(self, lst=[], foo=None):
...         self._lst, self.foo = 1st, foo
...     def append(self, item):
...         self._lst.append(item)
...     def __getitem__(self, item):
...         return self._lst[item]
...     def __setitem__(self, item, val):
...         self._lst [item] = val
...     def __delitem__(self, item):
...         del self._lst[item]
...
>>> foolst2 = oldFooList([1,2,3], 'F00')
>>> foolst2[1], foolst2.foo
(2, 'F00')
```

If you need more complex datatypes than the basic types, or even than an instance whose class has magic methods, often these can be constructed by using instances whose attributes are bound in link-like fashion to other instances. Such bindings can be constructed according to various topologies, including circular ones (such as for modeling graphs). As a simple example, you can construct a binary tree in Python using the following node class:

```
>>> class Node:
...     def __init__(self, left=None, value=None, right=None):
...         self.left, self.value, self.right = left, value, right
...     def __repr__(self):
...         return self.value
...
>>> tree = Node(Node(value="Left Leaf"),
...             "Tree Root",
...             Node(left=Node(value="RightLeft Leaf"),
...                  right=Node(value="RightRight Leaf") ))
>>> tree,tree.left,tree.left.left,tree.right.left,tree.right.right
(Tree Root, Left Leaf, None, RightLeft Leaf, RightRight Leaf)
```

In practice, you would probably bind intermediate nodes to names, in order to allow easy pruning and rearrangement.

SEE ALSO: int *18*; float *19*; list *28*; string *129*; tuple *28*; UserDict *24;* UserList *28*; UserString *33*;

A.4 Flow Control

Depending on how you count it, Python has about a half-dozen flow control mechanisms, which is much simpler than most programming languages. Fortunately, Python's collection of mechanisms is well chosen, with a high—but not obsessively high—degree of orthogonality between them.

From the point of view of this appendix, exception handling is mostly one of Python's flow control techniques. In a language like Java, an application is probably considered "happy" if it does not throw any exceptions at all, but Python programmers find exceptions less "exceptional"—a perfectly good design might exit a block of code *only* when an exception is raised.

Two additional aspects of the Python language are not usually introduced in terms of flow control, but nonetheless amount to such when considered abstractly. Both functional programming style operations on lists and Boolean shortcutting are, at the heart, flow control constructs.

A.4.1 if/then/else Statements

Choice between alternate code paths is generally performed with the if statement and its optional elif and else components. An if block is followed by zero or more elif blocks; at the end of the compound statement, zero or one else blocks occur. An if statement is followed by a Boolean expression and a colon. Each elif is likewise followed by a Boolean expression and colon. The else statement, if it occurs, has no Boolean expression after it, just a colon. Each statement introduces a block containing one or more statements (indented on the following lines or on the same line, after the colon).

Every expression in Python has a Boolean value, including every bare object name or literal. Any empty container (list, dict, tuple) is considered false; an empty string or Unicode string is false; the number 0 (of any numeric type) is false. As well, an instance whose class defines a .__nonzero__() or .__len__() method is false if these methods return a false value. Without these special methods, every instance is true. Much of the time, Boolean expressions consist of comparisons between objects, where comparisons actually evaluate to the canonical objects "0" or "1". Comparisons are <, >, ==, >=, <=, <>, !=, is, is not, in, and not in. Sometimes the unary operator not precedes such an expression.

Only one block in an "if/elif/else" compound statement is executed during any pass—if multiple conditions hold, the first one that evaluates as true is followed. For example:

```
>>> if 2+2 <= 4:
...   print "Happy math"
...
Happy math
>>> x = 3
>>> if x > 4: print "More than 4"
... elif x > 3: print "More than 3"
... elif x > 2: print "More than 2"
... else: print "2 or less"
...
More than 2
>>> if isinstance(2, int):
...     print "2 is an int"     # 2.2+ test
... else:
...     print "2 is not an int"
```

Python has no "switch" statement to compare one value with multiple candidate matches. Occasionally, the repetition of an expression being compared on multiple elif lines looks awkward. A "trick" in such a case is to use a dict as a pseudo-switch. The following are

equivalent, for example:

```
>>> if var.upper() == 'ONE':    val = 1
... elif var.upper() == 'TWO':   val = 2
... elif var.upper() == 'THREE': val = 3
... elif var.upper() == 'FOUR':  val = 4
... else:                        val = 0
...
>>> switch = {'ONE':1, 'TWO':2, 'THREE':3, 'FOUR':4}
>>> val = switch.get(var.upper(), 0)
```

A.4.2 Boolean Shortcutting

The Boolean operators or and and are "lazy." That is, an expression containingor or and evaluates only as far as it needs to determine the overall value. Specifically, if the first disjoin of an or is true, the value of that disjoin becomes the value of the expression, without evaluating the rest; if the first conjoin of an and is false, its value likewise becomes the value of the whole expression.

Shortcutting is formally sufficient for switching and is sometimes more readable and concise than "if/elif/else" blocks. For example:

```
>>> if this:         # 'if' compound statement
...     result = this
... elif that:
...     result = that
... else:
...     result = 0
...
>>> result = this or that or 0 # boolean shortcutting
```

Compound shortcutting is also possible, but not necessarily easy to read; for example:

```
>>> (cond1 and func1()) or (cond2 and func2()) or func3()
```

A.4.3 for/continue/break Statements

The for statement loops over the elements of a sequence. In Python 2.2+, looping utilizes an iterator object (which may not have a predetermined length)—but standard sequences like lists, tuples, and strings are automatically transformed to iterators in for statements. In earlier Python versions, a few special functions like xreadlines() and xrange() also act as iterators.

Each time a for statement loops, a sequence/iterator element is bound to the loop variable. The loop variable may be a tuple with named items, thereby creating bindings for multiple names in each loop. For example:

```
>>> for x,y,z in [(1,2,3),(4,5,6),(7,8,9)]: print x, y, z, '*',
...
1 2 3 * 4 5 6 * 7 8 9 *
```

A particularly common idiom for operating on each item in a dictionary is:

```
>>> for key,val in dct.items():
...     print key, val, '*',
...
1 2 * 3 4 * 5 6 *
```

When you wish to loop through a block a certain number of times, a common idiom is to use the range() or xrange() built-in functions to

create ad hoc sequences of the needed length. For example:

```
>>> for _ in range(10):
...     print "X",      # '_' is not used in body
...
X X X X X X X X X X
```

However, if you find yourself binding over a range just to repeat a block, this often indicates that you have not properly understood the loop. Usually repetition is a way of operating on a collection of related *things* that could instead be explicitly bound in the loop, not just a need to do exactly the same thing multiple times.

If the continue statement occurs in a for loop, the next loop iteration proceeds without executing later lines in the block. If the break statement occurs in a for loop, control passes past the loop without executing later lines (except the finally block if the break occurs in a try).

A.4.4 map(), filter(), reduce(), and List Comprehensions

Much like the for statement, the built-in functions *map(), filter()*, and *reduce()* perform actions based on a sequence of items. Unlike a for loop, these functions explicitly return a value resulting from this application to each item. Each of these three functional programming style functions accepts a function object as a first argument and sequence(s) as a subsequent argument(s).

The *map()* function returns a list of items of the same length as the input sequence, where each item in the result is a "transformation" of one item in the input. Where you explicitly want such transformed items, use of *map()* is often both more concise and clearer than an equivalent for loop; for example:

```
>>> nums = (1,2,3,4)
>>> str_nums = []
>>> for n in nums:
...     str_nums.append(str(n))
...
>>> str_nums
['1', '2', '3', '4']
>>> str_nums = map(str, nums)
>>> str_nums
['1', '2', '3', '4']
```

If the function argument of *map()* accepts (or can accept) multiple arguments, multiple sequences can be given as later arguments. If such multiple sequences are of different lengths, the shorter ones are padded with None values. The special value None may be given as the function argument, producing a sequence of tuples of elements from the argument sequences.

```
>>> nums = (1,2,3,4)
>>> def add(x, y):
...     if x is None: x=0
...     if y is None: y=0
...     return x+y
...
>>> map(add, nums, [5,5,5])
[6, 7, 8, 4]
>>> map(None, (1,2,3,4), [5,5,5])
[(1, 5), (2, 5), (3, 5), (4, None)]
```

The *filter()* function returns a list of those items in the input sequence that satisfy a condition given by the function argument. The function argument must accept one parameter, and its return value is interpreted as a Boolean (in the usual manner). For example:

```
>>> nums = (1,2,3,4)
>>> odds = filter(lambda n: n%2, nums)
>>> odds
(1, 3)
```

Both *map()* and *filter()* can use function arguments that have side effects, thereby making it possible—but not usually desirable—to replace every for loop with a *map()* or *filter()* function. For example:

```
>>> for x in seq:
...     # bunch of actions
...     pass
...
>>> def actions(x):
...     # same bunch of actions
...     return 0
...
>>> filter(actions, seq)
[]
```

Some epicycles are needed for the scoping of block variables and for break and continue statements. But as a general picture, it is worth being aware of the formal equivalence between these very different-seeming techniques.

The *reduce()* function takes as a function argument a function with two parameters. In addition to a sequence second argument, *reduce()* optionally accepts a third argument as an initializer. For each item in the input sequence, *reduce()* combines the previous aggregate result with the item, until the sequence is exhausted. While *reduce()*—like *map()* and *filter()*—has a loop-like effect of operating on every item in a sequence, its main purpose is to create some sort of aggregation, tally, or selection across indefinitely many items. For example:

```
>>> from operator import add
>>> sum = lambda seq: reduce(add, seq)
>>> sum([4,5,23,12])
44
>>> def tastes_better(x, y):
...     # some complex comparison of x, y
...     # either return x, or return y
...     # ...
...
>>> foods = [spam, eggs, bacon, toast]
>>> favorite = reduce(tastes_better, foods)
```

List comprehensions (listcomps) are a syntactic form that was introduced with Python 2.0. It is easiest to think of list comprehensions as a sort of cross between for loops and the *map()* or *filter()* functions. That is, like the functions, listcomps are expressions that produce lists of items, based on "input" sequences. But listcomps also use the keywords for and if that are familiar from statements. Moreover, it is typically much easier to read a compound list comprehension expression than it is to read corresponding nested *map()* and *filter()* functions.

For example, consider the following small problem: You have a list of numbers and a string of characters; you would like to construct a list of all pairs that consist of a number from the list and a character from the string, but only if the ASCII ordinal is larger than the number. In traditional imperative style, you might write:

```
>>> bigord_pairs = []
>>> for n in (95,100,105):
...     for c in 'aei':
...         if ord(c) > n:
...             bigord_pairs.append((n,c))
...
>>> bigord_pairs
[(95, 'a'), (95, 'e'), (95, 'i'), (100, 'e'), (100, 'i')]
```

In a functional programming style you might write the nearly unreadable:

```
>>> dupelms=lambda lst,n: reduce(lambda s,t:s+t,
...                              map(lambda l,n=n: [l]*n, 1st))
>>> combine=lambda xs,ys: map(None,xs*len(ys), dupelms(ys,len(xs)))
>>> bigord_pairs=lambda ns,cs: filter(lambda (n,c):ord(c)>n,
...                                   combine(ns,cs))
>>> bigord_pairs((95,100,105),'aei')
```

```
[(95, 'a'), (95, 'e'), (100, 'e'), (95, 'i'), (100, 'i')]
```

In defense of this FP approach, it has not *only* accomplished the task at hand, but also provided the general combinatorial function combine() along the way. But the code is still rather obfuscated.

List comprehensions let you write something that is both concise and clear:

```
>>> [(n,c) for n in (95,100,105) for c in 'aei' if ord(c)>n]
[(95, 'a'), (95, 'e'), (95, 'i'), (100, 'e'), (100, 'i')]
```

As long as you have listcomps available, you hardly *need* a general combine() function, since it just amounts to repeating the for clause in a listcomp.

Slightly more formally, a list comprehension consists of the following: (1) Surrounding square brackets (like a list constructor, which it is). (2) An expression that usually, but not by requirement, contains some names that get bound in the for clauses. (3) One or more for clauses that bind a name repeatedly (just like a for loop). (4) Zero or more if clauses that limit the results. Generally, but not by requirement, the if clauses contain some names that were bound by the for clauses.

List comprehensions may nest inside each other freely. Sometimes a for clause in a listcomp loops over a list that is defined by another listcomp; once in a while a nested listcomp is even used inside a listcomp's expression or if clauses. However, it is almost as easy to produce difficult-to-read code by excessively nesting listcomps as it is by nesting *map()* and *filter()* functions. Use caution and common sense about such nesting.

It is worth noting that list comprehensions are not as referentially transparent as functional programming style calls. Specifically, any names bound in for clauses remain bound in the enclosing scope (or global if the name is so declared). These side effects put a minor extra burden on you to choose distinctive or throwaway names for use in listcomps.

A.4.5 while/else/continue/break Statements

The while statement loops over a block as long as the expression after the while remains true. If an else block is used within a compound while statement, as soon as the expression becomes false, the else block is executed. The else block is chosen even if the while expression is initially false.

If the continue statement occurs in a while loop, the next loop iteration proceeds without executing later lines in the block. If the break statement occurs in a while loop, control passes past the loop without executing later lines (except the finally block if the break occurs in a try). If a break occurs in a while block, the else block is not executed.

If a while statement's expression is to go from being true to being false, typically some name in the expression will be re-bound within the while block. At times an expression will depend on an external condition, such as a file handle or a socket, or it may involve a call to a function whose Boolean value changes over invocations. However, probably the most common Python idiom for while statements is to rely on a break to terminate a block. Some examples:

```
>>> command = ''
>>> while command != 'exit':
...     command = raw_input('Command > ')
...     # if/elif block to dispatch on various commands
...
Command > someaction
Command > exit
>>> while socket.ready():
...     socket.getdata()  # do something with the socket
... else:
...     socket.close()    # cleanup (e.g. close socket)
...
>>> while 1:
...     command = raw_input('Command > ')
...     if command == 'exit': break
```