UnixWorld Online: Tutorial Article No. 005

The What, Why, Who, and Where of Python

Our Guido-like Python image was originally created by Nancy Wogsland, the Graphics/Multimedia Art Director for Digital Creations, was edited by the author, and is used with the permission of Paul Everitt at Digital Creations. We think it's wrapped around a can of spam.--Becca Thomas, Editor

Learn about Python, the language that wraps itself around a problem to squeeze out a solution, swallowing it whole.

By Aaron R. Watters

Python is an interpreted, object oriented, freely copyable programming language that may be used without fee in commercial products. It runs under several environments including many Unices, MS-Windows, OS/2, and Macintosh operating systems. It includes many modern programming language features together with many useful standard packages. Programmers may easily extend Python to interface to other arbitrary software components. Python may be used for fun, CGI scripts, system administration, code generation, graphical user interfaces, file-format conversions, and almost any other computational task, but the most exciting use of Python is for general software engineering and product development.

- First, a little Python
  - Tormenting friends and enemies
  - Spending other people's money
  - Primes, of course
- Geological forces at work
- What and where and who
- The Python core
  - Object orientation
  - Hashing and Dictionaries
  - It dices, it slices
  - A scope by any other name...

        - More junk mail
- Standard libraries, extensions and contributions
- Extending Python
    - Required extensions
    - Speeding things up
    - Reference counting
- A Python development strategy
    - During new development
    - During support and enhancement
- Why not Perl or FooLanguage Instead?
    - Tcl
    - Perl
    - Scheme
- An invitation

First, a Little Python

There are many sound, logical, objective reasons why Python is a good language. However, first I'd like to point out one unsound, illogical, and subjective reason I like Python--*it's fun*. To try to illustrate what a delight Python is, let me show you some examples.

Tormenting friends and enemies:
I maintain lists of email addresses of people who I occasionally torment with irritating messages. In the old days I used UNIX mail aliases to manage these lists, but now I use Python because it allows much greater flexibility. When I want to send a file to a list of victim addresses I use the following Python module, adorned with some comments set off by the ‘‘#’’ character:

```
# mailer module mailer.py
import posix # make posix system calls available

def mailit(filename, subject, list):
    # mail the file to each victim
    for victim in list:
        # make a shell mail command for this victim
        string = 'cat ' + filename + \
                 ' | mail -n -s ' + `subject` + ' ' + victim
        print string         # echo the command
        posix.system(string) # execute the command

usage = 'function: mailit(filename, subjectstring, list)'
```

This module defines the `mailit()` function, which iterates through a list of victim addresses, constructing a mail command to mail a file to each, and executing the commands in subprocesses.

First, the reader will note use of indentation in the source code. Python groups statements via indentation: a block of statements begins where the indentation increases a level and ends where the indentation returns to the previous level (for example below the `def` function definition and the `for` loop). As a consequence of this syntactic convenience the string construction line must be explicitly broken into two lines via a continuation mark `\`. I found this weird at first, but now I find it seductively appealing and addictive.

Second, users of shell scripting languages and other UNIX tools will note that all string constants in the code must be explicitly delimited via quotations, like `'cat '`, because Python is designed to be a general purpose language (descending from the Algol family) even though it is also a nice scripting tool. Overloaded addition explicitly concatenates string values such as:

```
'cat ' + filename + ' | mail -n -s ' + `subject` + ' ' + victim
```

where `filename`, `subject`, and `victim` are each function local variables with string values. The ''reverse quoting'' around `` `subject` `` converts the value of `subject` into a ''readable string representation.'' In general, with more interesting values such as dictionaries or lists or class instances, reverse quoting is a powerful and interesting tool, but in this particular case it just adds quote characters around the string value of the variable.

The `for` loop iterates through the elements of a sequence object (a list in this case), rather than a sequence of integers as in Pascal or C. In other examples the object of iteration could be a sequence of integers created via the builtin functions `range` or `xrange`.

The interactive use of this function looks something like this:

```
>>> from mailer import *
>>> print usage
function: mailit(filename, subjectstring, list)
>>> victims =['aaron', 'aaron@cs.rutgers.edu', 'aaron@hertz']
>>> mailit('mailer.py', 'more junk mail', victims)
cat mailer.py | mail -n -s 'more junk mail' aaron
cat mailer.py | mail -n -s 'more junk mail' aaron@cs.rutgers.edu
cat mailer.py | mail -n -s 'more junk mail' aaron@hertz
>>>
```

Although this example execution uses Python's ''interactive prompt'' interface, I usually use this function embedded in a Python based X-Windows graphical interface [See Figure 1]. Wouldn't you?

Spending other people's money

When I'm lucky I get to spend money that's not mine, but I usually have to tell someone how much I'm going to spend. Because arithmetic is one of my many weaknesses, I use Python to help me add up all my requests:

```
# summer.py
from string import split, atof # use these string conveniences

def calc(filename='orders'):
    f = open(filename,'r') # open the file
    text = f.read()        # read the whole file as a string
    f.close() # close the file
    list = split(text)     # put in a whitespace-separated substring list
    # now look for strings that start with $ and add them up, if possible
    total = 0.0
    for s in list:
        if s[0] == '$':
           try:
              total = total + atof(s[1:])
           except: pass    # s[1:] isn't a number, ignore it.
    return total
```

This module defines a function `calc()`, which reads the contents of a file and then looks for whitespace-separated substrings that start with a `$`, attempts to interpret the remainder of each such string as a number, and computes the sum of all such numbers. If any substring cannot be converted to a number the `atof()` conversion function will raise an error, which is caught by the `except` exception handler, with the result that the offending string will be ignored. The default file name is taken to be `orders`, but the value can be overridden by explicitly supplying an argument.

If I create a purchase request letter named `orders` that looks like:

```
Dear Sir:
   I don't want to spend too many of your $'s today, but
   I would like to spend $12 on a puppy from the ASPCA,
   $0.05 on a pencil and $1500 on a new Pentium Computer.

   Thanks ever so much!
```

Then I can interactively use the `calc()` function as follows:

```
>>> from summer import *
>>> calc()
1512.05
>>> # 1512.05 is the total!
```

If I want to make the interface ‘‘nicer,’’ I could wrap the function in a script that makes it look like a standard UNIX command, or I could put a nice graphical face on the thing--there are just so many options!

Primes, of course:
I wouldn't have a Ph.D. in computer science if I wasn't obsessively worried about determining lists of prime numbers. :-)

```
# primes.py: compute the list of primes <= Limit
# demonstrates the use of else in a for clause.
def PrimesLE(Limit):
    Primes = [2] # a list with the first prime
    counter = 3  # the next number
    while counter<=Limit:
           for KnownPrime in Primes:
               if counter % KnownPrime == 0:
                  # counter is divisible by a known prime...
                  break # abandon this one and try the next one
           else:
               # since we didn't break, counter is not divisible
               # by a previous prime... hence it's prime!
               Primes.append(counter)
           # advance the counter (but skip the even numbers).
           counter = counter + 2
    return Primes
```

Here we start with a list containing just the first prime (2), and iterate through the odd numbers up to the `Limit`, testing each against the elements of the current `Primes` list. If the current `counter` is divisible by a `KnownPrime` (*that is,* if:

```
counter % KnownPrime == 0
```

where `%` is the familiar C ‘‘modulo’’ operator) then we ignore it by breaking out of the for loop, and skipping the `else` clause. If a given value for `counter` is not divisible by any currently known prime, the loop will not break, and at the end of the loop the `else` clause will add the `counter` (which must be prime) to the end of the primes list using the list method `Primes.append(counter).`

Interactive use of the function might look like:

```
>>> import primes
>>> primes.PrimesLE(30)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

I hope these examples give you some taste of the fun you can have with Python, without even getting into the best features discussed further on: object oriented class structures, generalized dictionaries, graphical interfaces, network protocol support, etcetera. But programming for fun will only take you so far in life, so let’s get solemn and strategic for a while.

Geological Forces at Work

The amazing advances in hardware technology will drive existing methods of software development to extinction, or irrelevancy. This is nothing to be mourned, but programmers and software companies should prepared for the change and using the Python language can help. As the cost of hardware drops while the speed of hardware increases, software customers will demand products that take advantage of this speed to provide increased configurability, scriptability, and general flexibility. Traditional modes of software development will not meet this demand in a timely manner.

The hardware technician with a scope and a soldering iron is a rare bird these days, although they crowded the skies not so long ago. Similarly, software developers who write rigid, monolithic, stand-alone software systems will soon survive only in the shrinking preserves of legacy projects. Replacing the endangered traditional programmer are end users and lightly skilled neophytes who slap together simple, but beautiful applications using powerful scripting tools such as Visual Basic, PowerBuilder and even Perl, Awk, or Tcl (because they haven’t found Python yet). Also arising from the primordial muck are journeyman wizards who can use combinations of interpreted languages with compiled components to aide the neophytes and otherwise meet difficult requirements in powerful but simple ways.

You can use Python to transform yourself from the endangered species of programmer to the emerging wizard species. Software companies can also use Python to transform existing products into flexible, scriptable components, preparing those products to meet the demands of ever more demanding and sophisticated customers. This article hopes to help explain how.

What, Where, and Who

Python was developed and improved on primarily by Guido van Rossum, who named it after Monty Python’s Flying Circus. Initially Python was part of the Amoeba Project at CWI in the Netherlands. Guido released Python via Internet FTP distribution and continues to develop and improve the language to the gratification of an ever increasing audience of programmers and users.

The Python language descends from the Modula family of languages, except that it uses Lisp-like dynamic typing and borrows other features from other languages such as object orientation *ala* Smalltalk, functional programming extensions from FP, and conveniences from UNIX shell languages. One of the novel things about Python is that it doesn't contain anything new--every piece of Python descends from some feature of some other language that has been proven valuable over the years--but it offers all these useful features in a clean, simple, well-designed package, written in portable C.

The copyright permits nearly arbitrary use of the language and its source code, even for general commercial purposes: the only thing you can't do with Python is copyright it yourself or sue the authors for any problems with the package or its documentation. This flexibility makes Python amenable for use and modification as a component in commercial products. In particular, the Python copyright lacks the various commercial usage restrictions present in the GNU public license, for example. So, if you want to feather Python into your commercial product, with everything compiled (even the Python source, and with all Python-code modules byte compiled), and charge *mucho dinero* for it all, there's no problem.

CNRI Incorporated recently established the Python Software Activity (PSA), with Guido's active cooperation. The purpose of the PSA is to provide a source and clearing house for Python-related information, and to help promote the use and continued development of Python. The PSA Web site (http://www.python.org) is the starting point for all sorts of information about Python, including addresses for the central FTP site and various mirrors, Python documentation and publications, and pointers to other information sources such as mailing lists and archives, as well as information on current commercial applications of Python. Please see this web site or their anonymous FTP site (www.python.org or 132.151.1.76) for additional information.

Another excellent source of information on Python is the Python newsgroup (comp.lang.python), which includes periodic postings of the Python FAQ (Frequently Asked Questions). O'Reilly and Associates will be publishing a book on the language sometime Winter 1995-1996. In addition, the distribution comes with four books on using Python (a tutorial, a language reference, a libraries reference, and an extension programming reference) available in LaTeX, PostScript, HTML, Windows Help files, and other formats.

Python is ideal for rapid prototyping and development using the ‘‘scripting/extension’’ model. In this approach basic external access primitives and computationally intensive operations may be implemented as compiled extensions to Python, and high-level control can be implemented using Python scripts, to produce flexible, extensible, scriptable, rapidly developed software components that can be easily maintained and modified.

The Python Core

Python is petite, possessed of a highly modular design, and a small collection of very powerful orthogonal constructs that nonetheless allow elegant and concise expression of computational ideas.

Of course, as illustrated above, Python includes the standard iterators and control constructs we know and love: the conditional `if/elif/else`, the iterators `while/else` and `for/else` (each of which supports the dubious, but often useful `break` and `continue` constructs). As we have seen, the `else` clause of a loop executes at the end of a loop if the loop terminates normally--which is useful for iterations that intend to ‘‘look for something in a structure, and if it's there break out of the loop, else put it in the structure’’ among other places.

Python uses a termination model for named error handling where the the `raise` construct raises an error (oddly enough), the `try/except` construct is used to catch errors and `try/finally` is used to specify mandatory cleanup actions to be performed before exiting a block as the result of an error condition or even a `return`. For example:

```
f = open(filename, "w")
try:
   do_something_with(f)
finally:
   f.close()
```

Here the `try/finally` construct guarantees that the file will be closed under normal conditions, or even if `do_something_with(f)` raises a non-catastrophic error. If the function raises an error the `finally` clause will execute, closing the file and re-raising the error. Under certain catastrophic conditions (for instance, when someone switches off the machine, among other possibilities) a `finally` or `except` clause may not execute, however.

There are three basic ways to specify procedural abstraction: defining function ‘‘values’’ using `lambda`, defining a named function using `def`, or defining encapsulated methods within an object class definition. Arguments to functions are always passed ‘‘by value,’’ but a function can return a tuple of results that can be unpacked in a single assignment, as in this example:

```
>>> divmod(67,3)
(22, 1)
>>> (quotient, remainder) = divmod(100, 11)
>>> print quotient, remainder
9 1
```

Object classes and object orientation

Encapsulation of object classes is one of the more interesting and useful aspects of Python. The module given below defines four classes:

`QSroot`
: A ‘‘virtual superclass’’ that encapsulates common behaviors for the other classes (initialization, emptiness testing, and `Pop`)

`Stack`
: A class whose instances act as classical last-in, first-out object archivers

`Queue`
: A class whose instances act as classical first-in, first-out object archivers

`DoubleQ`
: A double-ended-queue class that allows additions and accesses to either the front or the back of the queue

Note that `Stack` and `Queue` instances receive common behaviors via inheritance from the `QSroot` class, and instances of `DoubleQ` inherit both stack and queue behaviors from the `Stack` and `Queue` classes. Internally, all instances of these classes use generalized dictionaries to store the items being archived.

```
# classes.py: simple demonstration of class definition and inheritance

class QSroot: # common behaviors superclass

   def __init__(self): # instance initializer
```

```
        self.front = self.back = None # no front or back initially
        self.store = {} # an empty generalized dictionary

    def isEmpty(self):  # emptiness testing method
        return self.front == None # if no front, it must be empty

    def Pop(self): # get/delete front element
        result = self.store[ self.front ] # get it
        del self.store[ self.front ] # delete it
        # reinitialize self, if this is the last element
        if self.front == self.back: self.__init__()
        else: self.front = self.front - 1 # otherwise decrement front
        return result

class Stack(QSroot):     # first-in/first-out archive
    def Push(self, item): # add new front
        # if structure is empty initialize front,back to 0
        if self.isEmpty(): self.front = self.back = 0
        else: self.front = self.front+1 # otherwise increment front
        self.store[self.front] = item # store the item at new front index

class Queue(QSroot): # last-in/first-out archive
    def Enqueue(self, item): # add new back, analogous to Stack.Push
        if self.isEmpty(): self.front = self.back = 0
        else: self.back = self.back-1
        self.store[self.back] = item

    GetFront = QSroot.Pop # a more appropriate name for a Queue method

# double queue, add ability to get/delete back element
class DoubleQ(Queue, Stack):
    def GetBack(self): # get/delete back element, analogous to QSroot.Pop
        result = self.store[ self.back ]
        del self.store[ self.back ]
        if self.front == self.back: self.__init__()
        else: self.back = self.back + 1
        return result
```

To create a `DoubleQ` interactively type:

```
>>> D = DoubleQ()
```

thus creating a structure that inherits all behaviors of the four classes. To put things into `D` use either `Enqueue` or `Push`:

```
>>> for c in 'Odd': D.Enqueue(c)
...
>>> for c in ' Example 1 ': D.Push(c)
...
```

(here the ‘‘...’’ ellipses indicates that the Python interactive parser needs a newline to recognize the end of the ‘‘for’’ loop). Finally, once `D` has contents, you can take out members using either `GetBack`, `Pop`, or `GetFront` (which is another name for `Pop`):

```
>>> try:
...     while 1: print D.GetBack(), D.GetFront()
... except KeyError: print "all done!"
...
d
```

```
d 1
O
  e
E l
x p
a m
all done!
```

Weird, huh?

The above example illustrates that Python supports object class definitions with method encapsulation and multiple inheritance. The class definition mechanism has many options and gives tremendous power to the programmer: you can even define objects that ‘‘look like’’ functions, numbers, lists, dictionaries or other fundamental Python types (or several of them at once). There is much more to be said about classes and object instances, but for the present I’ll just hope you are confused enough to look to the Python reference manuals and the copious distributed example code for more information.

Hashing and Dictionaries

Python offers many recent features like classes and such, but it also includes at least one extremely useful feature that is as old as the hills--or at least as old as a reasonably large tree (which is as old as anything gets in this industry)--hash-implemented generalized dictionaries. The notion of hashing and hash tables, beloved to Perl and Awk programmers and others, is built into the Python core language. The Python dictionary type allows the programmer to create efficient mappings between hashable Python objects and arbitrary values.

The simplest and probably the most common use for dictionaries is to map strings to objects, as in the following example. The `phone` module below defines a function that maps alphanumeric phone numbers such as `1-800-Fone-Sed` to strictly numeric representations such as `1-(800)-3663-733`.

```
from string import upper, joinfields

# a module 'constant' dictionary
keypad = { 'abc':2, 'def':3,
  'ghi':4, 'jkl':5, 'mno':6,
  'prs':7, 'tuv':8, 'wxy':9 }

# a derived module constant dictionary
letmap = {}
for (letters, number) in keypad.items():
    for letter in letters:
        letmap[letter] = letmap[upper(letter)] = `number`

# translate one letter
def transletter(letter):
    try: return letmap[letter]
    except KeyError: # not the fastest way, but it illustrates `in'...
         if letter in '0123456789-()': return letter
         else: raise ValueError, 'no translation for: '+`letter`

def translate(string):
    return joinfields( map( transletter, string ), '' )
```

Of course, this example illustrates a lot more than just dictionaries (such as the `map` function, which applies a function to each element of a sequence, producing a list of results), but for the present purpose we focus on the dictionaries `keypad` and `letmap`. These dictionaries are declared and populated at the time that the module is loaded (and only once, if the module is loaded more than once). The `keypad` dictionary is cribbed off my telephone, and defines which number is associated with which letter of the alphabet using the dictionary literal notation:

```
{ key1 : value1, key2 : value2, ... }
```

The derived dictionary `letmap` translates the mapping into a more usable form by iterating through the item pairs in `keypad` via the dictionary method `keypad.items()`--mapping each letter and its upper case incarnation individually to a string representation for the appropriate number. Thus, `letmap` can translate ‘‘H’’ and ‘‘Q’’ as follows:

```
>>> letmap['H']
'4'
>>> letmap['Q']
Traceback (innermost last):
  File "", line 1, in ?
KeyError: Q
```

where the last look-up raised a `KeyError` because someone at Ma Bell thought that no one would ever want to use a ‘‘Q’’ in a phone number. The remainder of the module defines two functions that make using `letmap` more convenient. The interactive use of the `translate` function looks like:

```
>>> translate('1(900)big-Pigs')
'1(900)244-7447'
```

More advanced uses of dictionaries may use more complex keys (the things mapped from) and more interesting values (the things mapped to).

However, Guido, in his wisdom, made sure that not all Python objects can be used as ‘‘keys’’ in a dictionary. More precisely, Python objects are divided into ‘‘mutables’’ (lists, dictionaries, tuples that recursively contain mutables, etcetera), which have internal representations that may be altered in place, and ‘‘immutables’’ (strings, numbers, tuples that recursively contain only other immutables, etcetera) that can never be altered, period. Only immutables are allowed to be used as dictionary keys because a hash table may never ‘‘find’’ a key that has mutated after it was installed in the table.

Well, actually, by using user defined classes you can get around this protection/restriction if you need to, at your own risk--see the reference manual. Guido, also in his wisdom, did not endeavor to make Python fool-proof, because, as programming lore states ‘‘fools are too clever.’’

Programmers can also use Python’s hashing strategy in advanced ways--for example, by combining hashing with Python’s archiving and external indexing facilities to build simple, persistent databases--but I digress, see the mention of `Dbm` and `pickle` below.

It dices, it slices

In addition to dictionaries, Python provides sequence objects (generally implemented as arrays) with various cute features. When I saw many of these features for the first time, I thought ‘‘that’s cute, but I’ll never use it.’’ Three days later, and henceforth, of course, I used them all the time. For example `list[-1]` gives the last item of a list, and if I want to shove in some values into a list between the third and fourth elements, I could type:

```
>>> list[3:3] = [0, -1, 'thirty']
```

Here `list[3:3]` refers to ‘‘the location just after `list[2]` but just before `list[3]`’’ and the ‘‘slice assignment’’ shoves in the elements of the right-hand list into that ‘‘location,’’ shifting all other elements as needed. Note, also that because lists are heterogeneous I may mix numbers and strings as elements of the `list`.

A scope by any other name...

Python uses lexical scoping with convenient modularity. Every python source code file automatically defines a module and all Top-level names are grouped into modules. Global names within a module may refer to classes, functions, other modules, or any other object. The `import` and `from` statements allow one module to refer to objects from another module’s namespace, with the difference that:

```
from Japan import Cars
```

adds the name `Cars` to the namespace of the current module as a reference to the external object, whereas:

```
import Japan
```

imports the module `Japan` as a local reference to the external module itself, allowing fully qualified references to, for example, `Japan.Cars` or any other object in `Japan`’s namespace. Classes in turn define a name space of class internals that may be methods or other class constants. Subclasses may override any internals of their superclasses. Class instances also define a ‘‘mini-namespace’’ of data slots.

A reference to *`instance.name`* will refer to a data slot of the instance, if there is one of that name, or otherwise will refer to the ‘‘nearest’’ internal name to the class of the instance in a left-most height-first search of the inheritance hierarchy (this is standard stuff in the object-oriented world, but what a mouthful!). Methods and functions, while they execute also have a namespace of local variables. An unqualified *`name`* always refers to a function or method local variable or, if there is no such local variable, a global name in the current module.

More junk mail

While you’re reeling from all that, let me have a little fun in the hope of illustrating some of these scoping concepts by generating a little junk mail.

```
# People.py, silly illustration of scoping.

# a module global form letter template, uses printf-like escapes...
Form = '''
%s %s %s:
Please come to my party for an introduction to the Python
programming language--entrance fee is only $300!  Please
bring %s and %s to share.
      %s,
         Aaron Watters
'''

# encapsulate information for sending a letter to a Person

class Person: # default to english bland behavior

   def __init__(self, name, gender, maritalstatus):
       # self, name, etcetera are method local variables
       self.name = name # assign local name to name slot in self instance
```

```
        self.status = (gender, maritalstatus)

    form = Form # default to English form above
    greeting = 'Dear' # this is a class constant
    signoff = 'Sincerely' # and another...
    salutation = {('male','married'):'Mister', ('male','single'):'Master',
                  ('female','single'):'Ms', ('female','married'):'Ms'}
    drink = {'female': 'fine wine', 'male': 'hard liquor'}
    eat = {'female': 'something sweet', 'male': 'meat'}

    def formletter(self):
        sex = self.status[0] # a method local variable
        # Use local self, slots, and class globals to generate a letter.
        # (the % operator substitutes the strings into the Form)
        print self.form % (
          self.greeting, self.salutation[self.status], self.name,
             self.eat[sex], self.drink[sex],
                self.signoff)

# spice things up for cowpersons, by shadowing some class globals...
class CowPerson(Person):
    greeting = 'Howdy'
    signoff = 'And watch out for those cow patties'
    drink = {'female': 'beer', 'male': 'moonshine'}
```

This example uses a global template for a form letter `Form` and defines two classes that encapsulate class constants which, for example, define the appropriate `greeting` for all members of that class. Thus, a `Person` will receive the bland treatment:

```
>>> Person('Willy','male','single').formletter()

Dear Master Willy:
Please come to my party for an introduction to the Python
programming language--entrance fee is only $300!  Please
bring meat and hard liquor to share.
      Sincerely,
         Aaron Watters
```

whereas a cowgirl recieves the more appropriate

```
CowPerson('SueBob', 'female', 'married').formletter()

Howdy Ms SueBob:
Please come to my party for an introduction to the Python
programming language--entrance fee is only $300!  Please
bring something sweet and beer to share.
      And watch out for those cow patties,
         Aaron Watters
```

Note that for a cowgirl instance `self.name` evaluates to an instance local value slot (the name for the person), `self.signoff` evaluates to the class global `CowPerson.signoff`, and `self.eat` evaluates to the inherited class global `Person.eat`.

Of course no good businessperson should ignore the international market:

```
# module latin.py, illustrates use of import
import People
```

```
# don’t use the form from module People... it’s in English.
Form = '''
%s %s %s:
Haga el favor de asistir en una fiesta para aprender
Python--entrada solamente $300!  Traiga %s
y %s para todos.
  %s,
     Aaron Watters
'''

# shadow all class globals to get Spanish behavior...
#  reference the external class People.Person...
class Spanish(People.Person):
   form = Form     # use the spanish form above
   greeting = 'Saludos'
   signoff = 'Muchas Gracias'
   salutation = {('male','married'):'SeNor', ('male','single'):'SeNor',
         ('female','single'):'SeNorita', ('female','married'):'SeNora'}
   drink = {'male': 'Aguardiente', 'female': 'sangria'}
   eat = {'male': 'algo picante', 'female': 'frutas frescas'}

class French(People.Person):
   def __init__(self, *args):
       raise SystemError, 'I forget my HS French!'
```

This example lets me to take advantage of the recent NAFTA agreement:

```
Spanish('Juanita', 'female', 'single').formletter()

Saludos SeNorita Juanita:
Haga el favor de asistir en una fiesta para aprender
Python--entrada solamente $300!  Traiga frutas frescas
y sangria para todos.
  Muchas Gracias,
     Aaron Watters
```

Here the `Spanish` class inherits only the methods `__init__` and `formletter` and overloads all other class global names from the `People` superclass. Because scoping is lexical, any non-local reference to `Form` in the `latin` module refers to `latin.Form` (whereas in Perl’s dynamic scoping, for example, some references might refer to `People.Form` depending on the state of the interpreter).

Modules, classes, instances, functions, and just about all other parts of Python are ‘‘first class objects,’’ which may be used as arguments to functions and examined dynamically. The run -time dynamic nature of Python objects is particularly convenient for testing, debugging, and troubleshooting.

The Python core language includes other features that cannot be explained here, but experienced programmers can pick up enough Python to use it productively in a day or so (no joke), and later they can look up additional features as they find the need.

Standard Libraries, Extensions, and Contributions

Python comes with an amazing collection of useful code, both implemented directly in Python or implemented as optional compiled extension modules (that may be loaded dynamically in many environments). Other libraries and extensions are available from Python contributors either from the Python FTP sites or from other archives. This topic deserves a small book, and it has one: a libraries manual automatically comes packaged into the Python distribution. Nevertheless, the discussion below briefly summarizes some of the libraries you get with Python, with emphasis on the ones I've used.

The Python libraries include a nice symbolic debugger and a profiler (both written in Python, of course), which help in identifying and fixing bugs and bottlenecks in Python code. There are also a number of other cute utilities, such as a program that automatically translates C preprocessor files that define constants into Python modules.

Operating system interfaces are provided both via a standard `os` package and via a `posix` module with related utilities. Library modules also provide support for basic network operations, such as modules that aid in sending and parsing Internet mail, transferring information using FTP, and class libraries for providing and receiving World Wide Web services.

Examples given here illustrate some of the functionality provided for manipulating strings and text in complex ways. Also included are facilities for matching and otherwise manipulating regular expressions. Python objects may also be translated to strings before being archived to the file system, or transferred to another process using the `marshal` and `pickle` modules. Encryption and decryption of strings is also supported via several alternative strategies. Interfaces to various indexing mechanisms (for example, `Dbm` allow strings--which may encode Python objects--to be archived efficiently in indexed files). Native interfaces to some commercial database systems (for example Oracle and Sybase) are also available.

In certain environments that support multi-threading Python allows the interpreter to run many threads at once. Platform specific modules also allow the manipulation of images and sound, among other things.

The graphical user interfaces (GUI) for Python provide some of Python's sexier features (and some would argue its naughty bits as well). The distribution includes an interface to the Tk graphical toolkit, which is portable to many UNIX platforms. Helpful enthusiasts have also contributed direct bindings to the X subroutine libraries, bindings to OSF/Motif, bindings to MicroSoft Foundation Classes (with UNIX translations provided via Tk), and a number of other GUI packages are mentioned in the Python FAQ. The GUI options for Python are exciting and useful, but also continue to evolve rapidly.

I necessarily omit much in this section. It behooves Python programmers to familiarize themselves with the Python libraries, extensions and contributions: they can speed up your coding considerably because you may be able to use them directly, or, at least, look to them for example usage of Python.

Extending Python

There are at least two reasons users may want to add compiled extension modules to those given in the Python distribution: necessity and speed.

Required Extensions
Complex applications may need to talk to existing subroutine libraries or special purpose devices that are not known to the Python distribution. For example, you may want Python to interact with your brand new fancy image scanner, or you may want to script linear programming subroutines purchased elsewhere from Python. In this case someone must explain to Python how to talk to these external interfaces via a compiled extension module; there is no alternative, unless somebody already did it--look through the distribution and contributed modules!

For example, if you need Python scripts to talk to an Oracle database you *could* write an extension module that allows Python to ‘‘call down’’ to Oracle’s native application programmer interface--*but DON’T DO IT!* Somebody already did! Check out the contributed modules at the FTP sites!

But, if you can’t find an existing interface, you’ll have to write one yourself. Happily, binding Python functions to external functionality is often a straightforward task aided by the many conveniences offered by the Python extension facilities: in the simplest case, basic accesses need only a thin wrapper of compiled functions that translate external data representations back and forth from the Python representations, calling the underlying accesses as needed. Although a complete example of such a module is beyond the scope of this presentation, I include a simple ‘‘hello world’’ extension module, which may be extended to provide external interfaces:

```
/* hello.c -- stupid example Python extension module in C. */
/* include a bunch of headers */
#include "allobjects.h"
#include "modsupport.h"
#include "ceval.h"
#ifdef STDC_HEADERS
#include <stddef.h>
#else
#include <sys/types.h>
#endif

/* define an external function, (stupid in this case) */
static object *sayHi(object *self, object *args)
{
   return newstringobject("Hi there!");
}

/* create a name binding structure including a ref to the function */
static struct methodlist thingy_methods[] = {
  {"sayHi",    (method)sayHi},
  {NULL, NULL}            /* sentinel */
};

/* create an initialization function for this module */
void
inithello()
{
  initmodule("thingy", thingy_methods);
}
```

To add this module to the Python executable, add one line to a configuration file in the Python source tree (in general, referencing any external libraries needed by the module) and type ‘‘make’’ at the top of the tree. Simple enough?

(Actually, if you get the HTML source for this article, you’ll have to replace the HTML escaped ‘‘less-than’’ and ‘‘greater-than’’ characters with the real versions too, but that’s not Guido’s fault.)

In many environments that provide dynamic linking, you can link a new module to the Python executable dynamically, as well.

It may happen that the desired external library uses structures that do not map easily to Python types, in which case the module can ‘‘Pythonize’’ the structure by defining a new compiled extension type that contains the structure, as discussed below.

Speeding things up

Alternatively, you may find that Python doesn't do what you want to do fast enough. In this case, you may wish to implement the part of the application that ''has to be fast'' as an extension module. As with almost any interpreted language, compute intensive applications implemented in Python will generally run an order of magnitude or more slower than a--hypothetical and much more difficult to implement or modify--implementation of the same algorithm in a compiled language. Alternatively, if your application is not compute intensive (such as network intensive, or user interface intensive, or database intensive applications) you may observe no noticeable difference in the speed between a pure Python implementation and a compiled analogue: so try Python first to get the bugs out and make sure there is actually a speed problem.

If the speed of the Python interpreter really is a problem, computationally intensive parts of the application may be sped up via a compiled extension module, but first look over your Python code to see if you can't improve it and eliminate the problem. If Python rewrites don't hack it then write your extension module, but please look through the Python distribution and the contributed modules first to make sure someone hasn't already implemented what you need.

For example, certain primitive image scanners operate on strings of four-bit nibbles that must be converted back and forth to other representations, such as genuine eight-bit strings. Although it is trivial to implement nibble operations directly using the Python core language, the result will be too slow for the manipulation of large images--so you *could* implement the required nibble operations as a compiled extension module to Python--*but DON'T DO IT!* Somebody already did: see the optional `imageop` module that comes with the Python distribution!

Once you've determined that a new Python extension is the only option you may find that writing Python extensions is easier and more fun than writing stand alone compiled applications. I did. It's trickier than writing Python, of course, and a buggy compiled extension can corrupt the rest of Python in arbitrary ways, but because Python was implemented from the start with extensions in mind, writing and testing Python compiled extension code is remarkably simple.

For one, if you've taken the advice given above, you already have a too slow, but working implementation of the functionality in Python, which has been analyzed, tested, and optimized. Now all you need to do is translate the algorithm into a--generally less terse and uglier--compiled language such as C. If the existing implementation uses special Python features you can selectively re-use those features by calling into the Python run-time library, for example to allow dictionary accesses, or even to ''call-back'' a function or method written in Python.

Furthermore, if your Python implementation uses an object class that logically maps to a C structure, you may re-implement the class as a new Python ''extension type.'' The treatment of basic Python types is one of the most beautiful aspects of the core implementation. To explain to Python how to manipulate a new type, you need to define an initialization function that creates the type, and a ''type structure'' that encapsulates the basic accesses and methods for the type. Each object of the new type must refer to the type structure as shown in this diagram. If you want, you can make your new type ''look like a number'' (or dictionary, or function, etcetera) to Python.

Reference counting

Compiled Python extensions do not benefit from the protection of the Python run-time system, of course, so many things must be handled with great care and discipline. Many of these gotchas are familiar to any journeyman programmer (dereferenced null pointers, references off the end of an array, use before initialization, missing `break`s in a `switch` construct, and the whole nasty clan of hoodlums), but Python contributes a new one: reference counts must be maintained correctly.

Python maintains a count of references to any Python object, which allows Python to deallocate the object once there are no references. Consequently, whenever an extension module creates a new reference to any object known to Python, the extension must increment the object's reference count, and conversely when the extension destroys a reference to an object, the extension must decrement the object's reference count. Upon testing, too few `INCREF`s manifest themselves as process crashes at apparently arbitrary times, which is distressing but easy to diagnose. Too few `DECREF`s generate a ''memory leak,'' which may manifest itself as an ever growing process size, which may not be noticeable until the module is used for large-scale applications, if ever.

In my case I've introduced reference count bugs by reasoning ''Gosh, if I don't INCREF it here, I won't have to DECREF it later--what a cool micro-optimization!'' (*BEEEP*. Next contestant please.) If you avoid such micro-optimization bugs, maintaining reference counts is pretty easy. Even considering the need to maintain reference counts, writing compiled Python extensions is, if anything, easier than other programming in compiled languages.

A Python Development Strategy

The software development process can be improved in many ways by incorporating Python into software systems:

During new development:
Every programmer knows that software users never know what they want until they see what they don't want. By developing entire application prototypes in Python, software engineers can rapidly present and modify possible functionality and design alternatives until the customers say they are happy. Then, in principle, the prototype can be re-implemented using a conventional compiled language.

However, once everyone has seen how nice it can be to use Python for application control, I suspect in many cases the Python component of the application will never be completely eliminated. After all, it may be the case that the Python implementation is perfectly acceptable, in which case, why do the rewrite?

Alternatively, if the prototype is too slow, critical components of the application could be rewritten as compiled extension modules to Python, primarily to increase execution speed, but the high-level logic of the application will continued to be managed by Python scripts. I would suggest using the following procedure (writing in pseudo-Python):

```
def Develop(vague_inaccurate_requirements):
    Product = a simple implementation of the requirements in Python.
    while the customers don't like the Product:
       Product = what the customers claim to want.
    while the product is too slow:
       Identify a bottleneck in the product by profiling.
       Optimize the bottleneck in Python.
       if it's still a bottleneck:
          Reimplement the bottleneck as a compiled extension module.
```

```
Develop an acceptance test in Python for the Product (in Python!).
while the Product doesn't pass the test:
   debug the Product (or the test!).
Deliver the Product with some or all Python code byte-compiled
   to protect proprietary source.
```

Of course, this procedure is somewhat simplistic. For example, it should include alpha/beta testing with accompanied revision iterations. By using a more advanced version of this multi-lingual hybrid strategy, developers can rapidly determine what the customer wants, and deliver that functionality in a suitable product.

Finally, the resulting software will have the delightful feature of including general purpose scripting facilities, which can be used to implement complex configuration options or other nice features, such as special purpose graphical interfaces.

During support and enhancement
Support of programs that consist of high-level Python control constructs combined with low-level compute intensive compiled extensions should be much simpler than supporting an analogous collection of monolithic applications. In many cases bugs or undesirable features may reside in the Python code, which will likely be easier to debug and fix, especially because debugging Python requires no recompilation or linking. Developers may provide simple enhancements by modifying existing Python modules or contributing additional Python modules--product modifications of this kind (that require no modifications to compiled extension modules) may be delivered over the Internet, without the need to recompile or reinstall any system component. Site specific enhancements may be provided by editing Python modules at the site, without the need for a complete development environment at the site.

Furthermore, by using Python's extensive archiving facilities, small amounts of Python code can be easily included with the product to aid in troubleshooting and support functions.

Why Not Perl or FooLanguage Instead?

Many of my claims for Python may also apply to other interpreted programming languages, such as Perl, Scheme, or others, but I believe Python is especially well suited for use in larger scale commercial applications. In justifying this claim I must try to sail carefully to avoid the rocky waters of nerd religious belief. I will try, but I will almost certainly fail, because I always do.

From Python's possible competitors we may immediately eliminate a large number. First Python may be freely copied and modified in source form, so we may eliminate any competitor that requires licensing or otherwise requires getting lawyers and contracts involved. Second, Python is *really free*, so we may eliminate any competitor with a ''free'' copyright that restricts commercial use of the code in any way.

At the end of this filter, a few possibilities trickle through:

Tcl
The Tcl language also provides scripting facilities with easy extension. The Tcl language is also why I found Python, because I didn't think that ''just strings'' was a sufficiently broad selection of basic data types. A deeper inspection of Tcl reveals a cute little language that does not include many standard programming language features that the modern programmer would expect to see in a modern language. Tcl scripts can be useful and efficient if they grow to no larger than a few pages, and if they manage a

relatively small amount of data. Beyond that limit, experience suggests looking elsewhere and in my view almost any application will eventually grow beyond those limits. Whoops, there goes my keel!

Perl

Perl 5 is a remarkably efficient, compact, and terse interpreted scripting language optimized for processing text files. Perl was written by the amazing Larry Wall (who, it sometimes seems, also wrote everything else). Afficionados of Sed and Awk and other UNIX tools often find Perl elegant, easy to read, and easy to modify, but many others don't. Below is a ''matrix multiplication'' function in Perl posted to a number of Internet newsgroups by Tom Christiansen:

```
sub mmult { my ($m1,$m2) = @_;
    my ($m1rows,$m1cols) = (scalar @$m1, scalar @{$m1->[0]});
    my ($m2rows,$m2cols) = (scalar @$m2, scalar @{$m2->[0]});
    unless ($m1cols == $m2rows) {  # raise exception, actually
        die "IndexError: matrices don't match: $m1cols != $m2rows";
    }
    my $result = [];
    my ($i, $j, $k);
    for $i (0 .. ($m1rows - 1 )) {
        for $j (0 .. ($m2cols - 1 )) {
            for $k ( 0 .. ($m1cols - 1)) {
                $result->[$i]->[$j] += $m1->[$i]->[$k] * $m2->[$k]->[$j];
            }
        }
    }
    return $result;
}
```

(By the way, I believe this is an excellent example of good Perl style.)

For comparison Roland Giersig translates the matrix multiplication function into Tcl as follows:

```
proc mmult {m1 m2} {
    set m2rows [llength $m2];
    set m2cols [llength [lindex $m2 0]];
    set m1rows [llength $m1];
    set m1cols [llength [lindex $m1 0]];
    if { $m1cols != $m2rows || $m1rows != $m2cols } {
        error "Matrix dimensions do not match!";
    }
    foreach row1 $m1 {
        set row {};
        for { set i 0 } { $i < $m2cols } { incr i } {
            set j 0;
            set element 0;
            foreach row2 $m2 {
                incr element [expr [lindex $row1 $j] * [lindex $row2 $i]];
                incr j;
            }
            lappend row $element;
        }
        lappend result $row;
    }
    return $result;
}
```

And here is a roughly analogous function in Python:

```
def mmult(m1,m2):
    m2rows,m2cols = len(m2),len(m2[0])
    m1rows,m1cols = len(m1),len(m1[0])
    if m1cols != m2rows: raise IndexError, "matrices don't match"
    result = [ None ] * m1rows
    for i in range( m1rows ):
        result[i] = [0] * m2cols
        for j in range( m2cols ):
           for k in range( m1cols ):
              result[i][j] = result[i][j] + m1[i][k] * m2[k][j]
    return result
```

This author feels that the Python version is easier on the eyes. When it comes to maintaining code (especially code written by others) or presenting scripting languages to unsophisticated end-users, aesthetics matters. And in matters of aesthetics I think Python wins compared to almost any other programming language.

Aesthetics aside, what's really important about Python is its portability and it ease of extension. Perl can be extended, but to extend Perl 5 you must master a handful of special purpose preprocessing tools, including a special purpose interface definition language. In contrast to extend Python you only need to learn its extension API, which may well be somewhat simpler than other APIs you may already know. Perl is also somewhat portable, but it is clearly designed with UNIX in mind, which makes portability for non-UNIX platforms problematic. For example, I was unable to locate any port of Perl 5 for the Macintosh, although one may appear any day now, apparently. The Python core language ports the Mac and other non-UNIX platforms now.

Scheme (of various flavors)
Scheme is a minimalist rendering of Lisp that is remarkably powerful and a favorite of certain programming language theorists and researchers. It descends from Lisp, however, and retains a syntax which is darn easy for computers, but many feel is difficult for humans to parse, read, and understand. To illustrate the syntax of Scheme I include a public domain rendering in Scheme of the ‘‘primes’’ example given earlier (in excellent Scheme style as well).

```
; primes By Ozan Yigit

(define  (interval-list m n)
  (if (> m n)
      '()
      (cons m (interval-list (+ 1 m) n))))

(define (sieve l)
  (define (remove-multiples n l)
    (if (null? l)
        '()
        (if  (= (modulo (car l) n) 0)      ; division test
             (remove-multiples n (cdr l))
             (cons (car l)
                   (remove-multiples n (cdr l))))))
  (if (null? l)
      '()
      (cons (car l)
            (sieve (remove-multiples (car l) (cdr l))))))
```

```
(define (primes<= n)
  (sieve (interval-list 2 n)))

; for example: (primes<= 300)
```

Scheme does not directly support such conveniences as object orientation or true multiple name space modularity. Nevertheless, if you want general purpose scripting/extension capabilities for an application, you probably can't get it in a smaller footprint than via the Elk or SIOD Scheme implementations, which are each portable, unencumbered, and designed for extension. For general software engineering and scripting, both for the implementor and for the user, I believe Python with its amiable conveniences and beautiful syntax, may be preferred.

An Invitation

At this point I'm floundering on the rocky coast. I hope that if you like Perl, Tcl, Scheme, or some other interpreted scripting/extension language I failed to mention that the critique above won't keep you from giving Python a try. Give it a day or two, or even a week; you may like it. It's a thing of beauty that you may find useful and profitable.

Edited by Becca Thomas / Online Editor / UnixWorld Online / beccat@wcmh.com

Go To Content   Search

Last Modified: Tuesday, 22-Aug-95 15:53:10 PDT