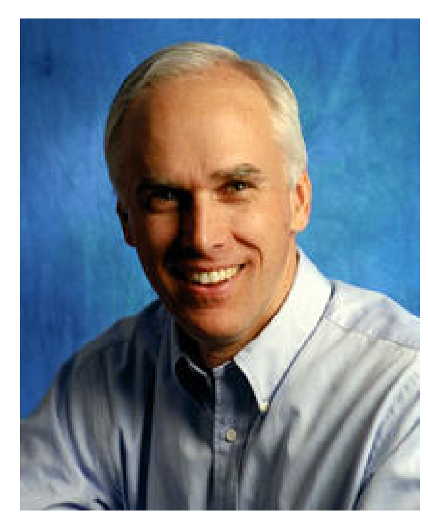
Rapid Application Development with



Scripting: Higher Level Programming for the 21st Century (IEEE Computer, March 1998)

http://home.pacbell.net/ouster/scripting.html

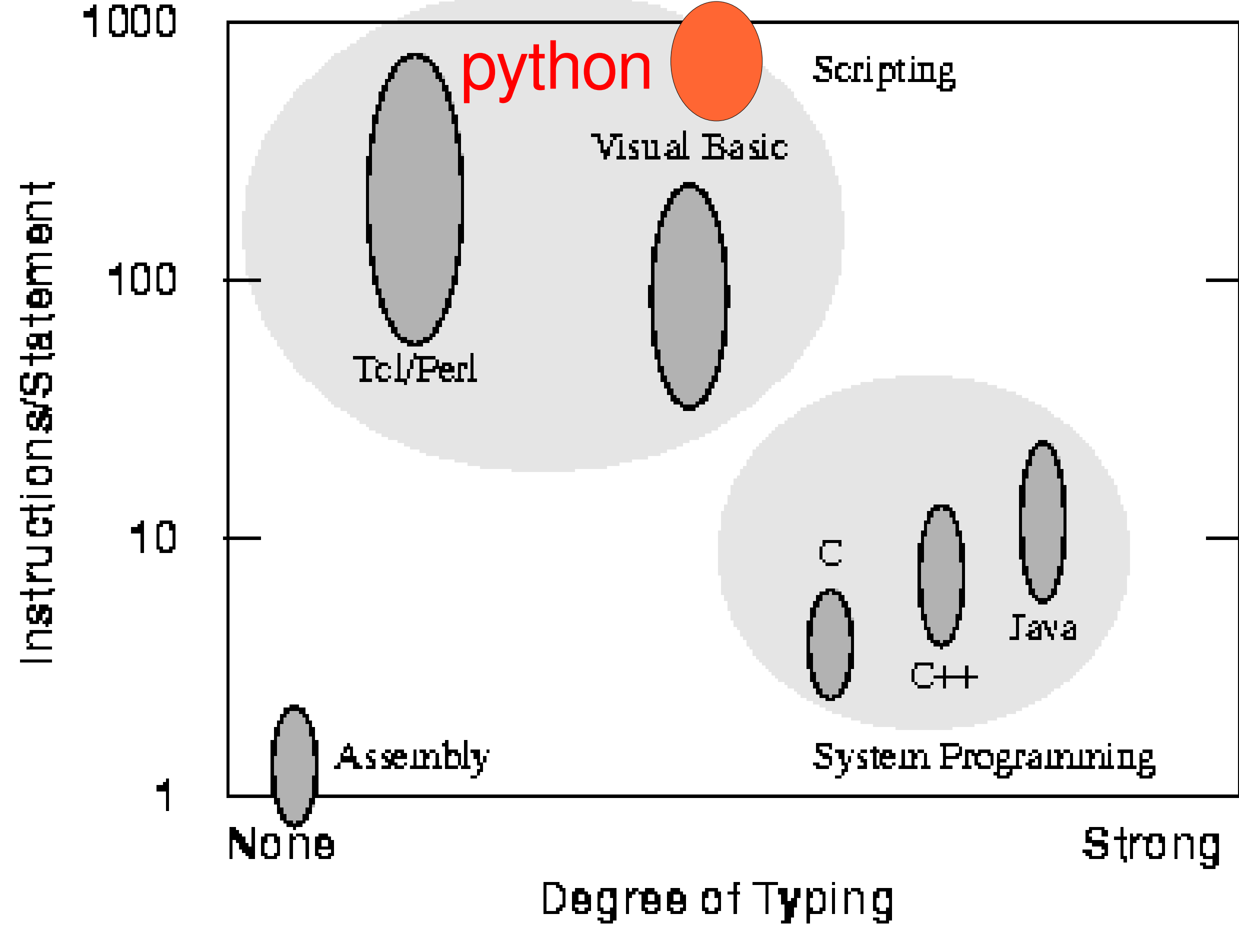


Figure 1. A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

Application (Contributor)	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months Tcl version: 1 day		60	C++ version implemented first; Tcl version had more functionality.
Computer system test and installation (Andy Belsey)	C test application: 272 Klines, 120 months. C FIS application: 90 Klines, 60 months. Tcl/Perl version: 7.7K lines, 8 months	47	22	C version implemented first. Tcl/Perl version replaced both C applications.
Database library (Ken Corey)	C++ version: 2-3 months Tcl version: 1 week		8-12	C++ version implemented first.
Security scanner (Jim Graham)	C version: 3000 lines Tcl version: 300 lines	10		C version implemented first. Tcl version had more functionality.
Display oil well production curves (Dan Schenck)	C version: 3 months Tcl version: 2 weeks		6	Tcl version implemented first.
Query dispatcher (Paul Healy)	C version: 1200 lines, 4-8 weeks Tcl version: 500 lines, 1 week	2.5	4-8	C version implemented first, uncommented. Tcl version had comments, more functionality.
Spreadsheet tool	C version: 1460 lines Tcl version: 380 lines	4		Tcl version implemented first.
Simulator and GUI (Randy Wang)	Java version: 3400 lines, 3-4 weeks. Tcl version: 1600 lines, < 1 week.	2	3-4	Tcl version had 10-20% more functionality, was implemented first.

Table 1. Each row of the table describes an application that was implemented twice, once with a system programming language such as C or Java and once with a scripting language such as Tcl. The **Code Ratio** column gives the ratio of lines of code for the two implementations (>1 means the system programming language required more lines); the **Effort Ratio** column gives the ratio of development times. In most cases the two versions were implemented by different people. The information in the table was provided by various Tcl developers in response to an article posted on the comp.lang.tcl newsgroup; see [7] for details.

Scripting Languages vs. System Programming Languages

- Boehm: programmer productivity LOC/day is **independent** of the language used ! → raise the **level of abstraction**: 1 LOC == many LOA.
- Different tasks:
  - System Programming: statically checked, **efficient**
  - Scripting: easy re-use (**glue-ing**) of powerful components.
- Characteristics of scripting languages:
  - **Interpreted** (no edit-compile-run cycle)
  - **Dynamic typing** (no need to declare, polymorphic)

"executable pseudo-code"

Scripting Languages vs. System Programming Languages

- Typical **software development process**:
  - Rapid Prototyping using scripting language
  - User feedback → new prototypes (eXtreme Programming)
  - Performance analysis → identify bottlenecks and replace by System Programming
- Extending and Embedding scripting languages
  - **Extending**: make external (library) code seemlessly seem as if part of the language
  - **Embedding**: System Programming Language application with script interpreter → user can extend.

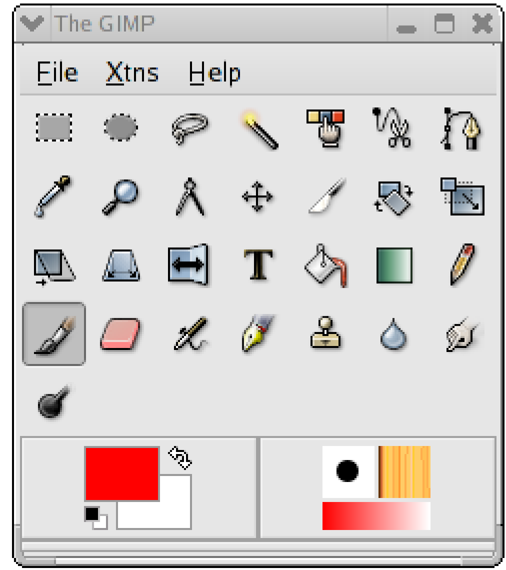
Extending

```
from Tkinter import *
root=Tk()
notice=Label(root, text="hello!")
notice.pack()
```

Embedding

Gimp application, Python-Fu

```
dir()
dir(gimp)
gimp.get_foreground()
gimp.set_foreground(255,0,0)
```



Python® is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.

Guido van Rossum

Standard Python documentation:

- Python website:
http://www.python.org

- Documentation index:
http://www.python.org/doc/2.4.2/

- Tutorial (recommended):
http://www.python.org/doc/2.4.2/tut/tut.html

- Language reference:
http://www.python.org/doc/2.4.2/ref/ref.html

- Standard library reference (very useful):
http://www.python.org/doc/2.4.2/lib/lib.html

Executing Python code:

- Python is (currently) **compiled into bytecode** which is then interpreted.

- You can put your Python scripts in plain text **files** (with extension .py) which can be executed from the command line as

```
python myscript.py
```

which generates the bytecode files (.pyc) for the file as well as all its dependencies.

- You can also experiment with Python from the top-level interpreter, where you can enter Python statements **interactively** and get the result immediately. The interpreter is "python" itself.

- Python comes with a simple IDE called IDLE (idle on Linux) but there are many other freely available IDEs.

- There are Python applications in all sorts of **domains**, from OS to 3D game engines, from scientific computing to databases.

The Python language

- Comments start with # or are enclosed in
  """ comment """

- strongly, dynamically typed, object-oriented,
  interpreted language.

- Scoping through indentation:

```
def incr(arg):
  return arg+1
```

- No type declarations, no variable declarations.
  Variables appear when first assigned, and their scope
  is the current code block:

```
x = 5
y = 'this is a string'
another = "this is also a string"
a_boolean = True
another_boolean = False
a_float = 3.5
```

Basic control structures:

```
if condition:
  statements

if condition:
  statements1
else:
  statements2

if condition1:
  statements1
elif condition2:
  statements2
else:
  statements

while condition:
  statements

for variable in sequence:
  statements
```

Indentation for defining nesting of code blocks:

```
if a:
  if b:
    print "one"
  else:
    print "two"
else:
  print "three"
```

Defining a function:

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n-1)
```

Imperative programming (has assignment and loops:)

```
def factorial(n):
  k = 1
  fact = 1
  while k <= n:
    fact = fact * k
    k = k + 1
  return fact
```

Default parameters: in the function declaration you can define the default value for a parameter:

```
def dummy(x = 1):
  print x

dummy(3)
```

>> prints 3

```
dummy()
```

>> prints 1

Functions can be declared **inside** functions

```
def difficult_task():
  def sub_task1():
    print "task 1"
  def sub_task2():
    print "task2"
  sub_task1()
  sub_task2()
```

Classes

```
class Robot:

  def __init__(self, x, y):  # This is the constructor
    self.x = x
    self.y = y
    self.dir = 0.0

  def turn(self, angle):
    self.dir = self.dir - angle

  def advance(self, distance):
    dx = distance * math.cos(self.dir)
    dy = distance * math.sin(self.dir)
    self.x = self.x + dx
    self.y = self.y + dx
```

(Note: 'self' is equivalent to 'this' in java, but all methods must specify it as parameter. You can call the variable 'self' anything you want.)

Objects (instantiation has the same syntax as function calling which can be very useful)

```
r = Robot(3,4)
```

Method calls

```
r.turn(math.pi)
r.advance(100)
```

Attribute access (all attributes are public by default)

```
print r.dir
```

Inheritance

```
class OneLeggedRobot(Robot):
  def hop(self):
    self.advance(100)
```

Built-in data-structures:

* **Tuples**:

```
trio = (3,'abc',True)
print trio[0]
print trio[2]
x,y,z = trio
print x
```

Tuples can be on the LHS of an assignment, resulting in 'parallel assignment', (this is called "unpacking.") For example, to swap two variables, you can do:

```
a, b = b, a
```

is shorthand for

```
(a, b) = (b, a)
```

Tuples are immutable (although their elements may be mutable.) This is, the following is illegal:

```
trio[1] = 'def'
```

*** Lists**

```
list = [3,'abc',True]
print list[0]
print list[2]
list = list + [4,'deg']
```

Lists, unlike tuples, can grow. Furthermore, unlike tuples, lists are mutable, so it is possible to do:

```
list[1] = 'def'
```

Equivalent:

```
i = 0
array = []
while i < 10:
  array.append(0)
  i = i + 1

array = []
for i in range(10):
  array = array + [0]

array = [0] * 10
```

*** Dictionaries**

```
dict = { 'key1': 'hello', 'key2': 'bonjour', 'key3': 'hola'}
print dict['key2']
dict['key4'] = 'bonjorno'
person = {'name':'Memo', 'age':32}
```

- Exception handling:

```
try:
  f(y)
  if cond:
    raise MyException("boo")
  g(z)
except MyException, arg:
  print "error:" + repr(arg)
```

- **Libraries**:

```
# Qualified access
import some_package
some_package.some_function()

# Unqualified access
from other_package import other_function
other_function()
```

Ernesto Posse is gratefully acknowledged for his "Python for programmers" tutorial from which most of this presentation is taken.