Introduction to Python

andrei.doncescu@laas.fr

Why Python?

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

- Have your cake and eat it, too: Productivity **and** readable code
- VHLLs will gain on system languages (John Ousterhout)
- "Life's better without braces" (Bruce Eckel)

For More Information?

http://python.org/

- documentation, tutorials, beginners guide, core distribution, ...

Books include:

- *Learning Python* by Mark Lutz
- *Python Essential Reference* by David Beazley
- *Python Cookbook*, ed. by Martelli, Ravenscroft and Ascher
- (online at http://code.activestate.com/recipes/langs/python/)
- http://wiki.python.org/moin/PythonBooks

4 Major Versions of Python

- “Python” or “CPython” is written in C/C++
  - Version 2.7 came out in mid-2010
  - Version 3.1.2 came out in early 2010

- “Jython” is written in Java for the JVM
- “IronPython” is written in C# for the .Net environment

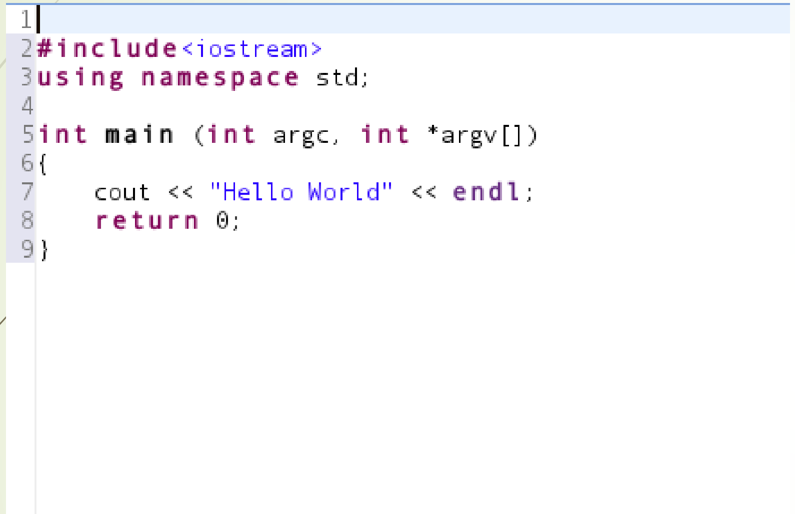
Go To Website

Lest's Start

- Installing Python
- Installing your text editor (NotePad++ or TextWrangler)
- Setting tab expansion
- Using the Command Line or Terminal Interface
- Editing and running Python Programs

Why Python?

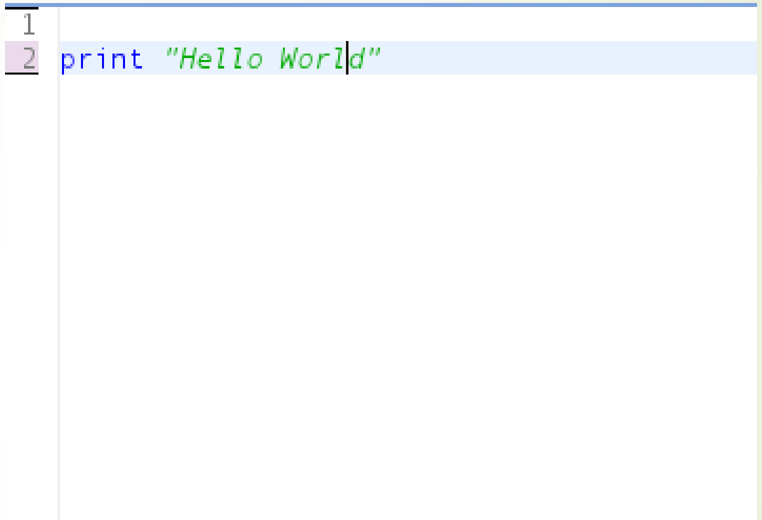
C++



```
#include<iostream>
using namespace std;

int main (int argc, int *argv[])
{
    cout << "Hello World" << endl;
    return 0;
}
```

- Complex syntax
- Difficult to read

Python



```

print "Hello World"
```

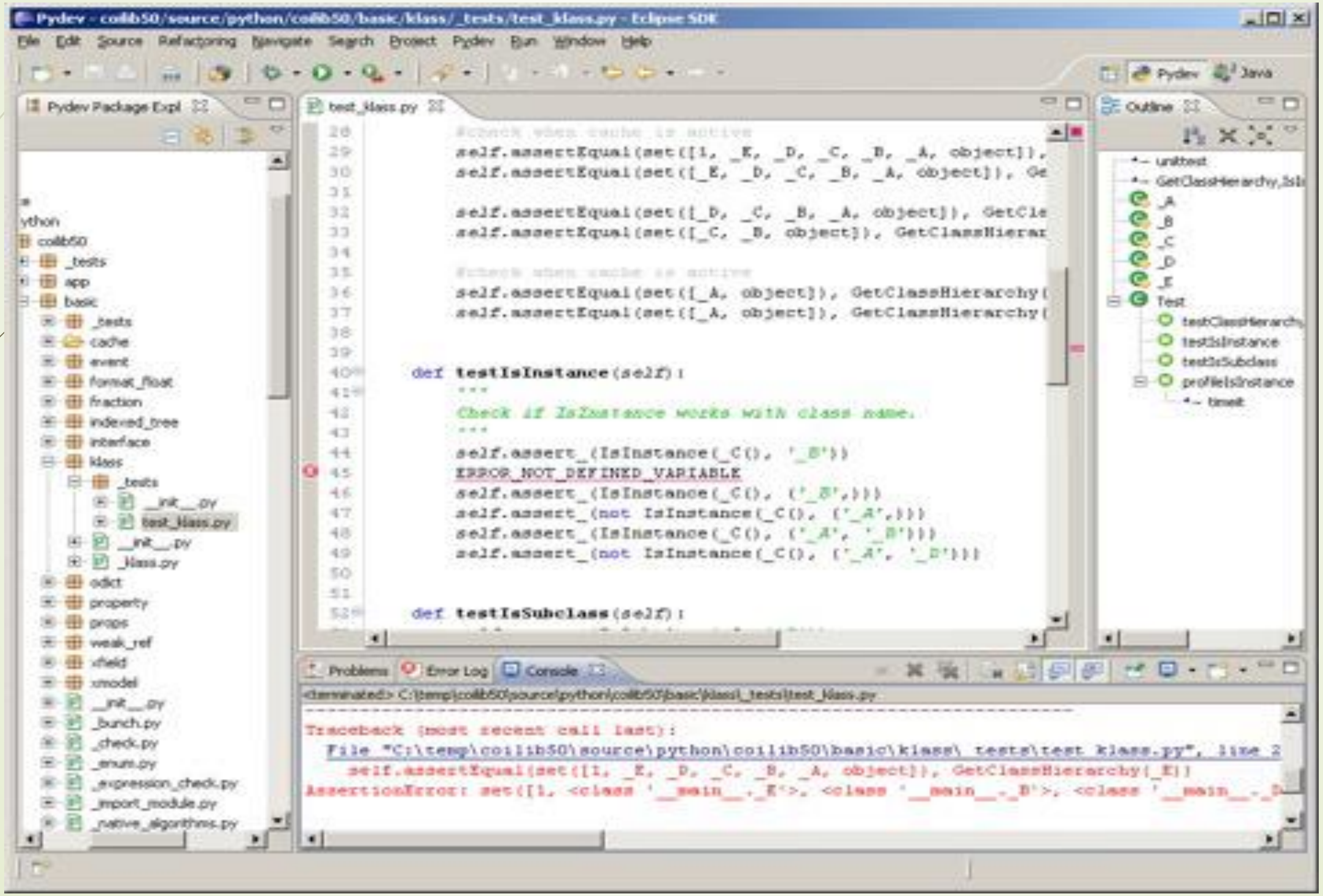
- Minimal Syntax
- Easier to read and debug
- Faster development time
- Increases productivity

Development Environments

what IDE to use? http://stackoverflow.com/questions/81584

1. PyDev with Eclipse   PyCharm
2. Komodo
3. Emacs
4. Vim
5. TextMate
6. Gedit
7. Idle
8. PIDA (Linux)(VIM Based)
9. NotePad++ (Windows)
10. BlueFish (Linux)

Pydev with Eclipse



```
        #check when cache is active
        self.assertEqual(set([1, _E, _D, _C, _B, _A, object]),
        self.assertEqual(set([_E, _D, _C, _B, _A, object]), Ge

        self.assertEqual(set([_D, _C, _B, _A, object]), GetCla
        self.assertEqual(set([_C, _B, object]), GetClassHierar

        #check when cache is active
        self.assertEqual(set([_A, object]), GetClassHierarchy(
        self.assertEqual(set([_A, object]), GetClassHierarchy(


    def testIsInstance(self):
        '''
        Check if IsInstance works with class name.
        '''
        self.assert_(IsInstance(_C(), '_B'))
        ERROR_NOT_DEFINED_VARIABLE
        self.assert_(IsInstance(_C(), ('_B',)))
        self.assert_(not IsInstance(_C(), ('_A',)))
        self.assert_(IsInstance(_C(), ('_A', '_B')))
        self.assert_(not IsInstance(_C(), ('_A', '_D')))


    def testIsSubclass(self):
```

```
<terminated> C:\temp\coilib50\source\python\coilib50\basic\klass\_tests\test_klass.py
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\temp\coilib50\source\python\coilib50\basic\klass\_tests\test_klass.py", line 2
    self.assertEqual(set([1, _E, _D, _C, _B, _A, object]), GetClassHierarchy(_E))
AssertionError: set([1, <class '__main__._E'>, <class '__main__._B'>, <class '__main__._D
```

Tutorial Outline

Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

Python is Interactive: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is a Beginner's Language: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules & packages
- exceptions
- files & standard library

Interactive “Shell”

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: IDLE (GUI),
  python (command line)
- Type statements or expressions at prompt:

```
>>> print "Hello, world"
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Interactive "Shell"

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: IDLE (GUI), python (command line)
- Type statements or expressions at prompt:

```
>>> print "Hello, world"
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

```
$sudo apt-get install python3-minimal
```

In the bash shell (Linux): type export PYTHONPATH=/usr/local/bin/python3.4 and press Enter.

Unix: IDLE is the very first Unix IDE for Python.

```
#!/usr/bin/python3

print ("Hello, Python!")
```

Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

  ```
  if friendly: greeting = "hello world"
  else: greeting = 12**2
  print greeting
  ```
- ***Everything*** is a "variable":
  - Even functions, classes, modules

User Input in Python

```
name = input("Give me your name: ")
print("Your name is " + name)
```

```
>>> Give me your name: Michel
Your name is Michel
```

What you get from the input() function is a string.
What can you do with it?

```
age = input("Enter your age: ")
age = int(age)
```

Do math with strings.

```
print("Were" + "wolf")
print("Door" + "man")
print("4" + "chan")
print(str(4) + "chan")
```

Numbers

- The usual suspects
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x | 1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
    - In Python 2.2 and beyond, 2**100 does the same thing
  - 1j**2 -> (-1+0j)

Python supports four different numerical types –

int (signed integers): They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

long (long integers): Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.

float (floating point real values) : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

complex (complex numbers) : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Grouping Indentation

In C:

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Conditionals: if statement

```
if age > 17:
  print("can see a rated R movie")
elif age < 17 and age > 12:
  print("can see a rated PG-13 movie")
else:
  print("can only see rated PG movies")
```

```
if a == 3:
  print("the variable has the value 3")
elif a != 3:
  print("the variable does not have the value 3")
```

```
if a == 3:
  print("the variable has the value 3")
else:
  print("the variable does not have the value 3")
```

If

- Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user.
- Hint: how does an even / odd number react differently when divided by 2?
- If the number is a multiple of 4, print out a different message.
- Ask the user for two numbers: one number to check (call it num) and one number to divide by (check). If check divides evenly into num, tell that to the user. If not, print a different appropriate message.

```
num = int(input("give me a number to check: "))
check = int(input("give me a number to divide by: "))

if num % 4 == 0:
    print(num, "is a multiple of 4")
elif num % 2 == 0:
    print(num, "is an even number")
else:
    print(num, "is an odd number")

if num % check == 0:
    print(num, "divides evenly by", check)
else:
    print(num, "does not divide evenly by", check)
```

Conditional

```
grade = input("Enter your grade: ")
if grade >= 90:
  print("A")
elif grade >= 80:
  print("B")
elif grade >= 70:
  print("C")
elif grade >= 65:
  print("D")
else:
  print("F")
```

50 ?

95?

Divisors

Create a program that asks the user for a number and then prints out a list of all the divisors of that number. (If you don’t know what a divisor is, it is a number that divides evenly into another number. For example, 13 is a divisor of 26 because 26 / 13 has no remainder.)

```
__author__ = ‘Bouty'

num = int(input("Please choose a number to divide: "))

listRange = list(range(1,num+1))

divisorList = []

for number in listRange:
    if num % number == 0:
        divisorList.append(number)

print(divisorList)
```

While Loops

The idea is simple: while a certain condition is True, keep doing something. For example:

```
a = 5
while (a > 0):
  print(a)
  a -= 1
```

The output of this code segment is:

```
5
4
3
2
1
```

A particularly useful way to use while loops is checking user input for correctness. For example:

```
quit = input('Type "enter" to quit:' )
while quit != "enter":
  quit = input('Type "enter" to quit:' )
```

Break Statement

A break statement stops the execution of a loop before the original condition is met. While the use of a break statement will often start an argument about good coding practices, sometimes it is useful.

For example:

```
 while True:
   usr_command = input("Enter your command: ")
   if usr_command == "quit":
     break
   else:
     print("You typed " + usr_command)
```

In this case, the break statement is used to break off the “infinite while loop” that we have constructed with the while True statement.

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation"""      # optional doc string
    statements

return                  # from procedure
return expression            # from function
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Function: Make a two-player Rock-Paper-Scissors game

Remember the rules:
Rock beats scissors
Scissors beats paper
Paper beats rock

```
import sys

user1 = input("What's your name?")
user2 = input("And your name?")
user1_answer = input("%s, do yo want to choose rock, paper or scissors?" % user1)
user2_answer = input("%s, do you want to choose rock, paper or scissors?" % user2)

def compare(u1, u2):
    if u1 == u2:
        return("It's a tie!")
    elif u1 == 'rock':
        if u2 == 'scissors':
            return("Rock wins!")
        else:
            return("Paper wins!")
    elif u1 == 'scissors':
        if u2 == 'paper':
            return("Scissors win!")
        else:
            return("Rock wins!")
    elif u1 == 'paper':
        if u2 == 'rock':
            return("Paper wins!")
        else:
            return("Scissors win!")
    else:
        return("Invalid input! You have not entered rock, paper or scissors, try again.")
        sys.exit()

print(compare(user1_answer, user2_answer))
```

Lists

- Flexible arrays, *not* Lisp-like linked lists
    - a = [99, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
    - a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
    - a[0] = 98
    - a[1:2] = ["bottles", "of", "beer"]

      -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
    - del a[-1]   # -> [98, "bottles", "of", "beer"]

LIST

Take two lists, say for example these two:

```
a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

and write a program that returns a list that contains only the elements that are common between the lists (without duplicates). Make sure your program works on two lists of different sizes.

One of the interesting things you can do with lists in Python is figure out whether something is inside the list or not. For example:

```
>>> a = [5, 10, 15, 20]
>>> 10 in a
True
>>> 3 in a
False
```

You can of course use this in loops, conditionals, and any other programming constructs.

```
list_of_students = ["Michele", "Sara", "Cassie"]

name = input("Type name to check: ")
if name in list_of_students:
  print("This student is enrolled.")
```

List Indexing

- In Python (and most programming in general), you start counting lists from the number 0. The first element in a list is "number 0", the second is "number 1", etc.
- As a result, when you want to get single elements out of a list, you can ask a list for that number element:

```
>>> a = [5, 10, 15, 20, 25]
>>> a[3]
20
>>> a[0]
5
```

- There is also a convenient way to get sublists between two indices:

```
>>> a = [5, 10, 15, 20, 25, 30, 35, 40]
>>> a[1:4]
```

- [10, 15, 20]

```
>>> a[6:]
[35, 40]
>>> a[:-1]
[5, 10, 15, 20, 25, 30, 35]
```

- The first number is the "start index" and the last number is the "end index."
- You can also include a third number in the indexing, to count how often you should read from the list:

```
>>> a = [5, 10, 15, 20, 25, 30, 35, 40]
>>> a[1:5:2]
[10, 20]
>>> a[3:0:-1] so   [15, 10, 5]
```

To read the whole list, just use the variable name (in the above examples, a), or you can also use [:] at the end of the variable name (in the above examples, a[:]).

Strings are lists

Because strings are lists, you can do to strings everything that you do to lists.
You can iterate through them:

```
string = "example"
for c in string:
  print "one letter: " + c
one letter: e
one letter: x
one letter: a
one letter: m
one letter: p
one letter: l
one letter: e
```

You can take sublists:

```
>>> string = "example"
>>> s = string[0:5]
>>> print s
exam
```

Strings

- "hello"+"world" "helloworld" # concatenation
- "hello"*3 "hellohellohello" # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ell" # slicing
- len("hello") 5 # size
- "hello" < "jello"1 # comparison
- "e" in "hello" 1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	

List

- A collection allows us to put many values in a single “variable”
- A collection is nice because we can carry all many values around in one convenient package.

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]

carryon = [ 'socks', 'shirt', 'perfume' ]
```

What is not a "Collection"

- Most of our variables have one value in them - when we put a new value in the variable - the old value is over written

```
$ python
Python 2.5.2 (r252:60911, Feb 22 2008, 07:57:53)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
>>> x = 2
>>> x = 4
>>> print x
4
```

List Constants

- List constants are surrounded by square brakets and the elements in the list are separated by commas.
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print [1, 24, 76]
[1, 24, 76]
>>> print ['red', 'yellow', 'blue']
['red', 'yellow', 'blue']
>>> print ['red', 24, 98.6]
['red', 24, 98.599999999999994]
>>> print [ 1, [5, 6], 7]
[1, [5, 6], 7]
>>> print []
[]
```

Display List Elements

```
for i in [5, 4, 3, 2, 1] :
    print i
print 'Blastoff!'
```

```
5
4
3
2
1
Blastoff!
```

Lists and definite loops - best pals

```
friends = ['Joseph', 'Gully', 'Sally']
for friend in friends :
    print 'Happy New Year:',  friend
print 'Done!'
```

```
Happy New Year: Joseph
Happy New Year: Gully
Happy New Year: SallyDone!
```

Looking Inside Lists

- Just like strings, we can get at any single element in a list using an index specified in square brackets

Joseph	Glenn	Sally
0	1	2

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print friends[1]
Glenn
>>>
```

Lists are Mutable

- Strings are "immutable" - we *cannot* change the contents of a string - we must make a new string to make any change
- Lists are "mutable" - we *can* change an element of a list using the index operator

```
>>> fruit = 'Banana’
>>> fruit[0] = 'b’
Traceback
TypeError: 'str' object does not
support item assignment
>>> x = fruit.lower()
>>> print x
banana
>>> lotto = [2, 14, 26, 41, 63]
>>> print lotto[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print lotto
[2, 14, 28, 41, 63]
```

How Long is a List?

- The len() function takes a list as a parameter and returns the number of *elements* in the list
- Actually len() tells us the number of elements of *any* set or sequence (i.e. such as a string...)

```
>>> greet = 'Hello Bob’
>>> print len(greet)
9
>>> x = [ 1, 2, 'joe', 99]
>>> print len(x)
4
>>>
```

Using the range function

- The range function returns a list of numbers that range from zero to one less than the parameter
- We can construct an index loop using for and an integer iterator

```
>>> print range(4)
[0, 1, 2, 3]
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print len(friends)
3
>>> print range(len(friends))
[0, 1, 2]
>>>
```

A tale of two loops...

```
friends = ['Joseph', 'Glenn', 'Sally']

for friend in friends :
    print 'Happy New Year:',  friend

for i in range(len(friends)) :
    friend = friends[i]
    print 'Happy New Year:',  friend
```

```
>>> friends = ['Joseph', 'Glenn', 'Sally']
>>> print len(friends)
3
>>> print range(len(friends))
[0, 1, 2]
>>>
```

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
```

Concatenating lists using +

- We can create a new list by adding two exsiting lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
>>> print a
[1, 2, 3]
```

Lists can be sliced using :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41,12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

Remember: *Just like in strings*, the second number is "up to but not including"

List Methods

```
>>> x = list()
>>> type(x)<type 'list'>
>>> dir(x)['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>>
```

http://docs.python.org/tutorial/datastructures.html

Building a list from scratch

- We can create an empty list and then add elements using the append method
- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print stuff
['book', 99]
>>> stuff.append('cookie')
>>> print stuff
['book', 99, 'cookie']
```

Is Something in a List?

- Python provides two operators that let you check if an item is in a list
- These are logical operators that return True or False
- They do not modify the list

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
>>>
```

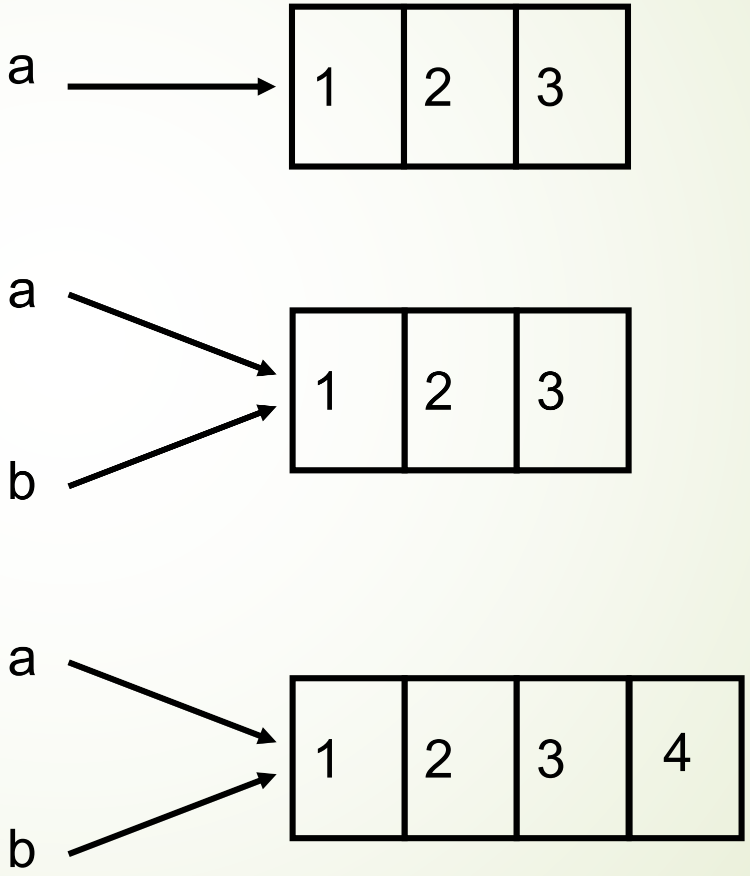
A List is an Ordered Sequence

- A list can hold many items and keeps those items in the order until we do something to change the order
- A list can be sorted (i.e. change its order)
- The sort method (unlike in strings) means "sort yourself"

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print friends
['Glenn', 'Joseph', 'Sally']
>>> print friends[1]
Joseph>>>
```

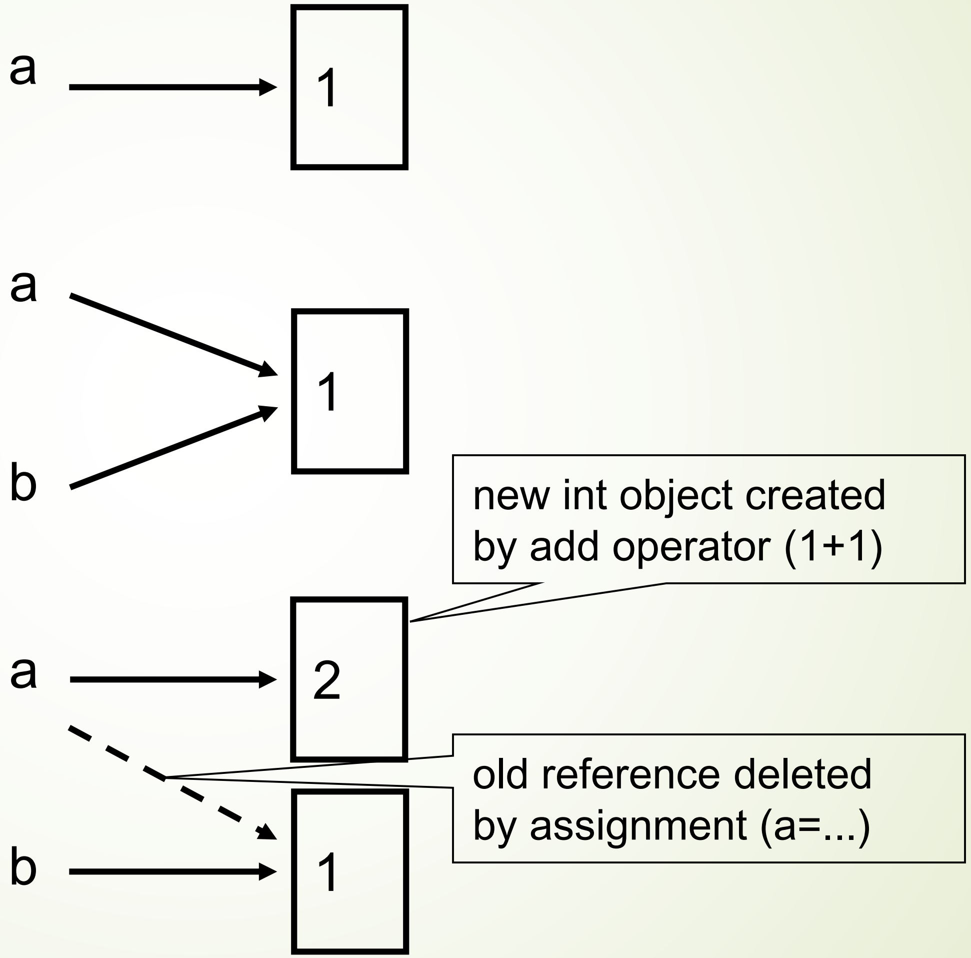
Changing a Shared List

```
a = [1, 2, 3]

b = a

a.append(4)
```



Changing an Integer

```
a = 1

b = a

a = a+1
```



new int object created by add operator (1+1)

old reference deleted by assignment (a=...)

Built in Functions and Lists

- There are a number of functions built into Python that take lists as parameters
- Remember the loops we built? These are much simpler

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

http://docs.python.org/lib/built-in-funcs.html

```
total = 0
count = 0
while True :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print 'Average:', average
```

```
Enter a number: 3
Enter a number: 9
Enter a number: 5
Enter a number: done
Average: 5.66666666667
```

```
numlist = list()
while True :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

Best Friends: Strings and Lists

```
>>> abc = 'With three words’
>>> stuff = abc.split()
>>> print stuff
['With', 'three', 'words']
>>> print len(stuff)
3
>>> print stuff[0]
With
```

```
>>> print stuff
['With', 'three', 'words']
>>> for w in stuff :
...        print w
...
 With
Three
Words
>>>
```

Split breaks a string into parts produces a list of strings. We think of these as words. We can access a particular word or loop through all the words.

```
>>> line = 'A lot               of spaces’
>>> etc = line.split()
>>> print etc['A', 'lot', 'of', 'spaces']
>>>
>>> line = 'first;second;third’
>>> thing = line.split()
>>> print thing['first;second;third']
>>> print len(thing)
1
>>> thing = line.split(';')
>>> print thing['first', 'second', 'third']
>>> print len(thing)
3
>>>
```

When you do not specify a delimiter, multiple spaces are treated like “one” delimiter.

You can specify what delimiter character to use in the splitting.

More List Operations

```
>>> a = [0,1,2,3,4] ;
>>> a.append(5)        # [0,1,2,3,4,5]
>>> a.pop()            # [0,1,2,3,4]
5
>>> a.insert(0, 42)    # [42,0,1,2,3,4]
>>> a.pop(0)           # [0,1,2,3,4]
5.5
>>> a.reverse()        # [4,3,2,1,0]
>>> a.sort()           # [0,1,2,3,4]
```

- The method append() appends a passed obj into the existing list.

- Syntax
- Following is the syntax for append() method −
- list.append(obj)

- Parameters
- obj -- This is the object to be appended in the list.

- Return Value
- This method does not return any value but updates existing list.

- Example
- The following example shows the usage of append() method.

- #!/usr/bin/python

- aList = [123, 'xyz', 'zara', 'abc'];
- aList.append(2009);
- print "Updated List : ", aList
- When we run above program, it produces following result −

- Updated List : [123, 'xyz', 'zara', 'abc', 2009]

Dictionaries

- Hash tables, "associative arrays"
  - d = {"duck": "eend", "water": "water"}
- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - del d["water"] # {"duck": "eend", "back": "rug"}
  - d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - d["duck"] = "duik" # {"duck": "duik", "back": "rug"}

More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

- key = (lastname, firstname)
- point = x, y, z # parentheses optional
- x, y, z = point   # unpack
- lastname = key[0]
- singleton = (1,)   # trailing comma!!!
- empty = ()   # parentheses!
- tuples vs. lists; tuples immutable

Reference Semantics

- Assignment manipulates references
    - x = y **does not make a copy** of y
    - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

What is an Object?

- A software item that contains variables and methods
- Object Oriented Design focuses on
  - Encapsulation:
    - dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - the ability to create subclasses that contain specializations of their parents

Namespaces

- At the simplest level, classes are simply namespaces
  - class myfunctions:
  -     def exp():
  -         return 0

  - >>> math.exp(1)
  - 2.71828...
  - >>> myfunctions.exp(1)
  - 0
- It can sometimes be useful to put groups of functions in their own namespace to differentiate these functions from other similarly named ones.

Python Classes

- Python contains classes that define objects
  - Objects are instances of classes

__init__ is the default constructor

- class atom:
-     def __init__(self,atno,x,y,z):
-         self.atno = atno
-         self.position = (x,y,z)

self refers to the object itself,
like *this* in Java.

Example: Atom class

```
class atom:
    def __init__(self,atno,x,y,z):
            self.atno = atno
            self.position = (x,y,z)
    def symbol(self):   # a class method
            return Atno_to_Symbol[atno]
    def __repr__(self): # overloads printing
            return '%d %10.4f %10.4f %10.4f' %
                    (self.atno, self.position[0],
                     self.position[1],self.position[2])
```

```
>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

Atom class

- Overloaded the default constructor
- Defined class variables (atno,position) that are persistent and local to the atom object
- Good way to manage shared memory:
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner programs result
- Overloaded the print operator

- We now want to use the atom class to build molecules...

Molecule Class

```
class molecule:
    def __init__(self,name='Generic'):
        self.name = name
        self.atomlist = []
    def addatom(self,atom):
        self.atomlist.append(atom)
    def __repr__(self):
        str = 'This is a molecule named %s\n' % self.name
        str = str+'It has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            str = str + `atom` + '\n'
        return str
```

Using Molecule Class

```
>>> mol = molecule('Water')
>>> at = atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print mol
This is a molecule named Water
It has 3 atoms
8  0.000 0.000 0.000
1  0.000 0.000 1.000
1  0.000 1.000 0.000
```

- Note that the print function calls the atoms print function
  - Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have one place to update.

Inheritance

```
class qm_molecule(molecule):
    def addbasis(self):
        self.basis = []
        for atom in self.atomlist:
            self.basis = add_bf(atom,self.basis)
```

- __init__, __repr__, and __addatom__ are taken from the parent class (molecule)
- Added a new function addbasis() to add a basis set
- Another example of code reuse
  - Basic functions don't have to be retyped, just inherited
  - Less to rewrite when specifications change

Overloading parent functions

```
class qm_molecule(molecule):
    def __repr__(self):
        str = 'QM Rules!\n'
        for atom in self.atomlist:
            str = str + `atom` + '\n'
        return str
```

- Now we only inherit __init__ and addatom from the parent
- We define a new version of __repr__ specially for QM

Adding to parent functions

- Sometimes you want to extend, rather than replace, the parent functions.

```
class qm_molecule(molecule):
    def __init__(self,name="Generic",basis="6-31G**"):
        self.basis = basis
        molecule.__init__(self,name)
```

add additional functionality
to the constructor

call the constructor
for the parent function

Public and Private Data

- Currently everything in atom/molecule is public, thus we could do something really stupid like
    - >>> at = atom(6,0.,0.,0.)
    - >>> at.position = 'Grape Jelly'

  that would break any function that used at.poisition
- We therefore need to protect the at.position and provide accessors to this data
  - Encapsulation or Data Hiding
  - accessors are "gettors" and "settors"
- Encapsulation is particularly important when other people use your class

Public and Private Data, Cont.

- In Python anything with two leading underscores is private
    - __a, __my_variable
- Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.
    - _b
  - Sometimes useful as an intermediate step to making data private

Encapsulated Atom

```
class atom:
    def __init__(self,atno,x,y,z):
        self.atno = atno
        self.__position = (x,y,z)  #position is private
    def getposition(self):
        return self.__position
    def setposition(self,x,y,z):
        self.__position = (x,y,z)  #typecheck first!
    def translate(self,x,y,z):
        x0,y0,z0 = self.__position
        self.__position = (x0+x,y0+y,z0+z)
```

Why Encapsulate?

- By defining a specific interface you can keep other modules from doing anything incorrect to your data
- By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users
  - Write to the Interface, not the the Implementation
  - Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

Classes that look like arrays

- Overload __getitem__(self,index) to make a class act like an array

```
class molecule:
    def __getitem__(self,index):
            return self.atomlist[index]

>>> mol = molecule('Water')  #defined as before
>>> for atom in mol:       #use like a list!
            print atom
>>> mol[0].translate(1.,1.,1.)
```

- Previous lectures defined molecules to be arrays of atoms.
- This allows us to use the same routines, but using the molecule class instead of the old arrays.
- An example of focusing on the interface!

Classes that look like functions

- Overload __call__(self,arg) to make a class behave like a function

```
class gaussian:
    def __init__(self,exponent):
            self.exponent = exponent
    def __call__(self,arg):
            return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.)
>>> func(3.)
0.0001234
```

Other things to overload

- __setitem__(self,index,value)
  - Another function for making a class look like an array/dictionary
  - a[index] = value
- __add__(self,other)
  - Overload the "+" operator
  - molecule = molecule + atom
- __mul__(self,number)
  - Overload the "*" operator
  - zeros = 3*[0]
- __getattr__(self,name)
  - Overload attribute calls
  - We could have done atom.symbol() this way

Other things to overload, cont.

- __del__(self)
  - Overload the default destructor
  - del temp_atom
- __len__(self)
  - Overload the len() command
  - natoms = len(mol)
- __getslice__(self,low,high)
  - Overload slicing
  - glycine = protein[0:9]
- __cmp__(self,other):
  - On comparisons (<, ==, etc.) returns -1, 0, or 1, like C's strcmp

References

- *Design Patterns: Elements of Reusable Object-Oriented Software,* Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (The Gang of Four) (Addison Wesley, 1994)
- *Refactoring: Improving the Design of Existing Code*, Martin Fowler (Addison Wesley, 1999)
- *Programming Python*, Mark Lutz (ORA, 1996).

Classes

class *name*:

"*documentation*"

statements

-or-

class *name*(*base1*, *base2*, ...):

...

Most, *statements* are method definitions:

def *name*(self, *arg1*, *arg2*, ...):

...

May also be *class variable* assignments

Example Class

```
class Stack:
   "A well-known data structure…"
   def __init__(self):          # constructor
      self.items = []
   def push(self, x):
      self.items.append(x)        # the sky is the limit
   def pop(self):
      x = self.items[-1]              # what happens if it’s empty?
      del self.items[-1]
      return x
   def empty(self):
      return len(self.items) == 0 # Boolean result
```

Using Classes

- To create an instance, simply call the class object:

```
x = Stack() # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()  # -> 1
x.push(1)                    # [1]
x.empty()  # -> 0
x.push("hello")                     # [1, "hello"]
x.pop()              # -> "hello"        # [1]
```

- To inspect instance variables, use dot notation:

```
x.items              # -> [1]
```

Subclassing

```
class FancyStack(Stack):
    "stack with added ability to inspect inferior stack items"

    def peek(self, n):
        "peek(0) returns top; peek(-1) returns item below that; etc."
        size = len(self.items)
        assert 0 <= n < size                  # test precondition
        return self.items[size-1-n]
```

Subclassing (2)

```
class LimitedStack(FancyStack):
    "fancy stack with limit on stack size"

    def __init__(self, limit):
        self.limit = limit
        FancyStack.__init__(self)          # base class constructor

    def push(self, x):
        assert len(self.items) < self.limit
        FancyStack.push(self, x)           # "super" method call
```

Class / Instance Variables

```
class Connection:
    verbose = 0                   # class variable
    def __init__(self, host):
        self.host = host          # instance variable
    def debug(self, v):
        self.verbose = v          # make instance variable!
    def connect(self):
        if self.verbose:          # class or instance variable?
            print "connecting to", self.host
```

Instance Variable Rules

- On use via instance (self.x), search order:
    - (1) instance, (2) class, (3) base classes
    - this also works for method lookup
- On assignment via instance (self.x = ...):
    - always makes an instance variable
- Class variables "default" for instance variables
- But...!
    - mutable *class* variable: one copy *shared* by all
    - mutable *instance* variable: each instance its own

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables
- Importing modules:
  - import re; print re.match("[a-z]+", s)
  - from re import match; print match("[a-z]+", s)
- Import with rename:
  - import re as regex
  - from re import match as m
  - Before Python 2.0:
    - import re; regex = re; del re

Packages

- Collection of modules in directory
- Must have __init__.py file
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo()   # new

Catching Exceptions

```
def foo(x):
   return 1/x

def bar(x):
   try:
      print foo(x)
   except ZeroDivisionError, message:
      print "Can’t divide by zero:", message

bar(0)
```

Try-finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()       # always executed
print "OK"# executed on success only
```

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range"
- try:
      something
  except: # catch everything
      print "Oops"
      raise   # reraise

More on Exceptions

- User-defined exceptions
  - subclass Exception or any other standard exception
- Old Python: exceptions can be strings
  - WATCH OUT: compared by object identity, not ==
- Last caught exception info:
  - sys.exc_info() == (exc_type, exc_value, exc_traceback)
- Last uncaught exception (traceback printed):
  - sys.last_type, sys.last_value, sys.last_traceback
- Printing exceptions: traceback module

File Objects

- f = open(*filename*[, *mode*[, *buffersize*])
  - mode can be "r", "w", "a" (like C stdio); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - buffersize: 0=unbuffered; 1=line-buffered; buffered
- methods:
  - read([*nbytes*]), readline(), readlines()
  - write(*string*), writelines(*list*)
  - seek(*pos*[, *how*]), tell()
  - flush(), close()
  - fileno()

Standard Library

- Core:
  - os, sys, string, getopt, StringIO, struct, pickle, ...
- Regular expressions:
  - re module; Perl-5 style patterns and matching rules
- Internet:
  - socket, rfc822, httplib, htmllib, ftplib, smtplib, ...
- Miscellaneous:
  - pdb (debugger), profile+pstats
  - Tkinter (Tcl/Tk interface), audio, *dbm, ...

URLs

- http://www.python.org
  - official site
- http://starship.python.net
  - Community
- http://www.python.org/psa/bookstore/
  - (alias for http://www.amk.ca/bookstore/)
  - Python Bookstore

Further Reading

- Learning Python: Lutz, Ascher (O'Reilly '98)
- Python Essential Reference: Beazley (New Riders '99)
- Programming Python, 2nd Ed.: Lutz (O'Reilly '01)
- Core Python Programming: Chun (Prentice-Hall '00)
- The Quick Python Book: Harms, McDonald (Manning '99)
- The Standard Python Library: Lundh (O'Reilly '01)
- Python and Tkinter Programming: Grayson (Manning '00)
- Python Programming on Win32: Hammond, Robinson (O'Reilly '00)
- Learn to Program Using Python: Gauld (Addison-W. '00)
- And many more titles...

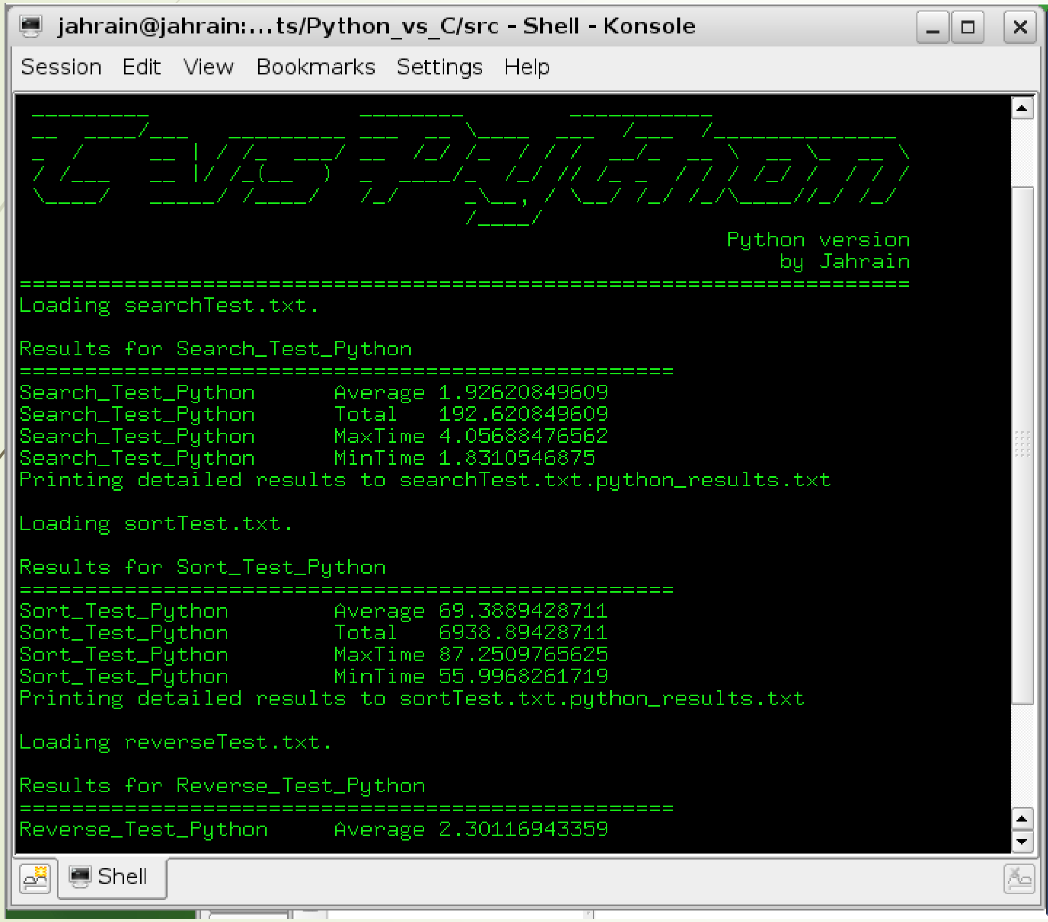
- Write performance benchmarking software in both C/C++ and Python.
- Execute software on Subaru's Real Time System (AO188RTS)
- Construct a Wiki page on Subaru's Wiki to present an analysis on the results.

Compiled vs. Interpreted

- C++ is a compiled language.
- Code is translated from a human readable text form into an executable form that a machine can read.
- Compiled code is hardware specific.

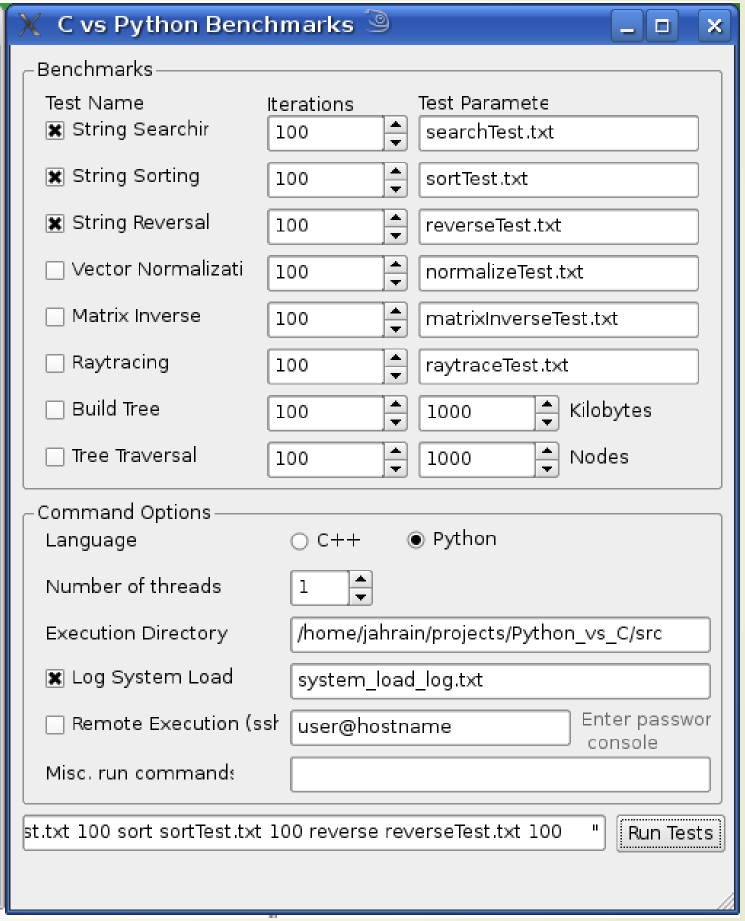
- Python is an interpreted language.
- Code is translated into a machine readable form during run time by an interpreter application.
- Interpreted code run on any platform with the interpreter installed.

Benchmarking Suite



```
                                                      Python version
                                                          by Jahrain
====================================================================
Loading searchTest.txt.

Results for Search_Test_Python
==================================================
Search_Test_Python       Average 1.92620849609
Search_Test_Python       Total   192.620849609
Search_Test_Python       MaxTime 4.05688476562
Search_Test_Python       MinTime 1.8310546875
Printing detailed results to searchTest.txt.python_results.txt

Loading sortTest.txt.

Results for Sort_Test_Python
==================================================
Sort_Test_Python         Average 69.3889428711
Sort_Test_Python         Total   6938.89428711
Sort_Test_Python         MaxTime 87.2509765625
Sort_Test_Python         MinTime 55.9968261719
Printing detailed results to sortTest.txt.python_results.txt

Loading reverseTest.txt.

Results for Reverse_Test_Python
==================================================
Reverse_Test_Python      Average 2.30116943359
```

Output



Test Name	Iterations	Test Paramete
☒ String Searchir	100	searchTest.txt
☒ String Sorting	100	sortTest.txt
☒ String Reversal	100	reverseTest.txt
☐ Vector Normalizati	100	normalizeTest.txt
☐ Matrix Inverse	100	matrixInverseTest.txt
☐ Raytracing	100	raytraceTest.txt
☐ Build Tree	100	1000 Kilobytes
☐ Tree Traversal	100	1000 Nodes

Language ○ C++ ● Python

Number of threads 1

Execution Directory /home/jahrain/projects/Python_vs_C/src

☒ Log System Load system_load_log.txt

☐ Remote Execution (ssh user@hostname Enter passwor console

Misc. run commands

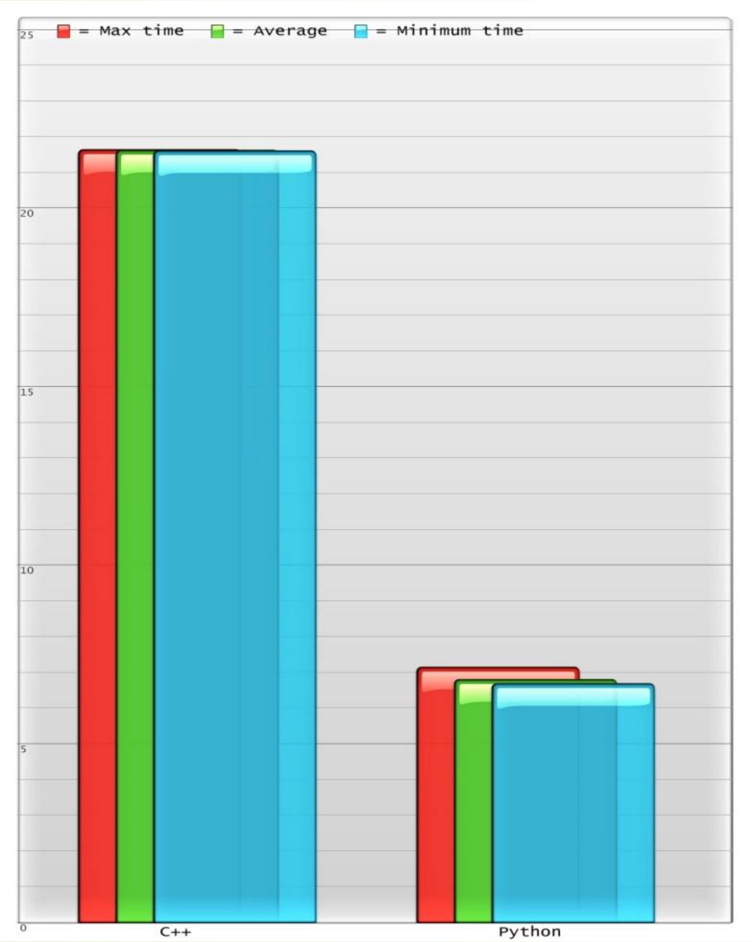
st.txt 100 sort sortTest.txt 100 reverse reverseTest.txt 100 "

GUI

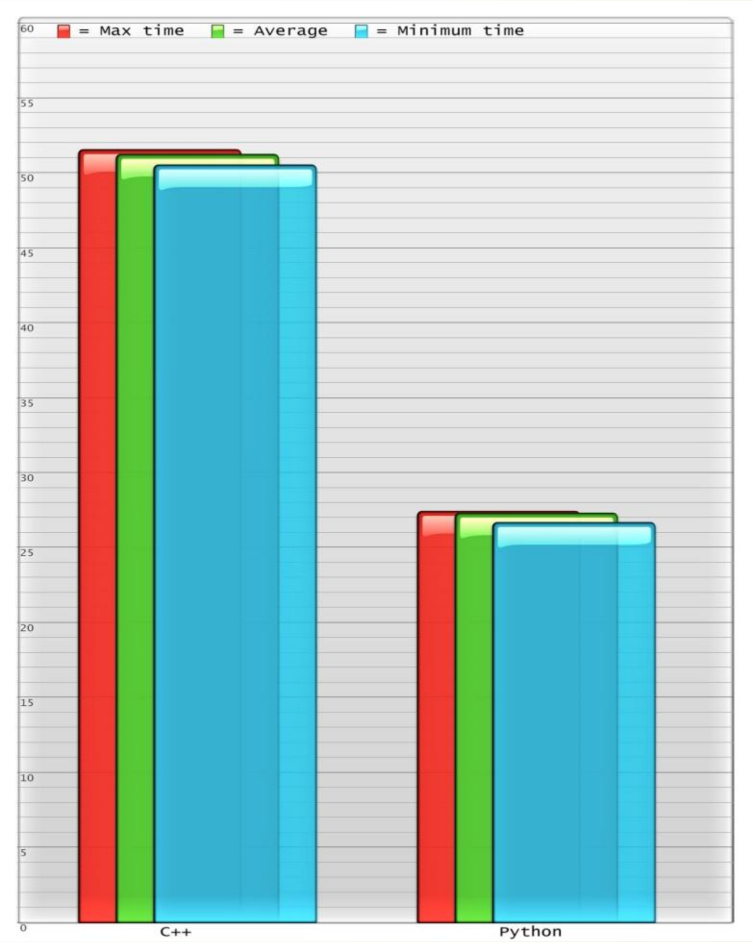
(Shorter is better)

Results - The Good

Searching



Sorting

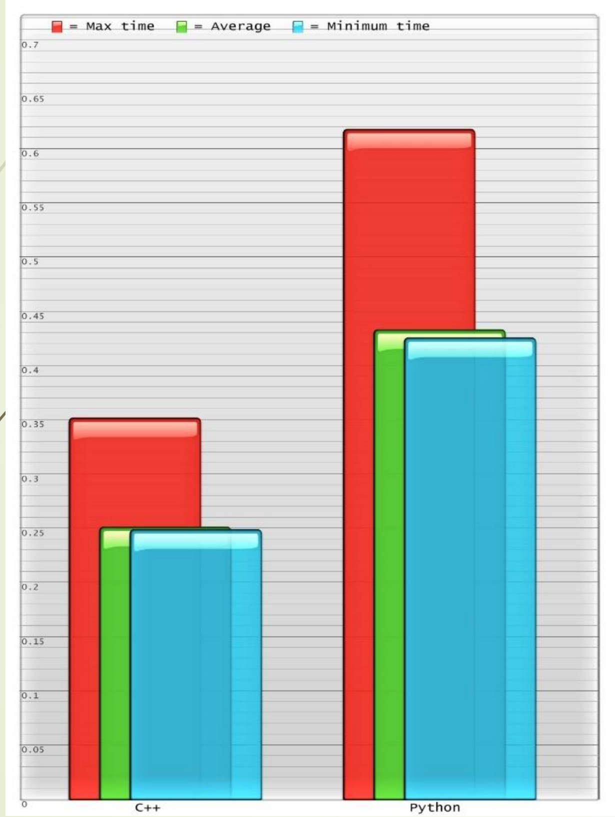


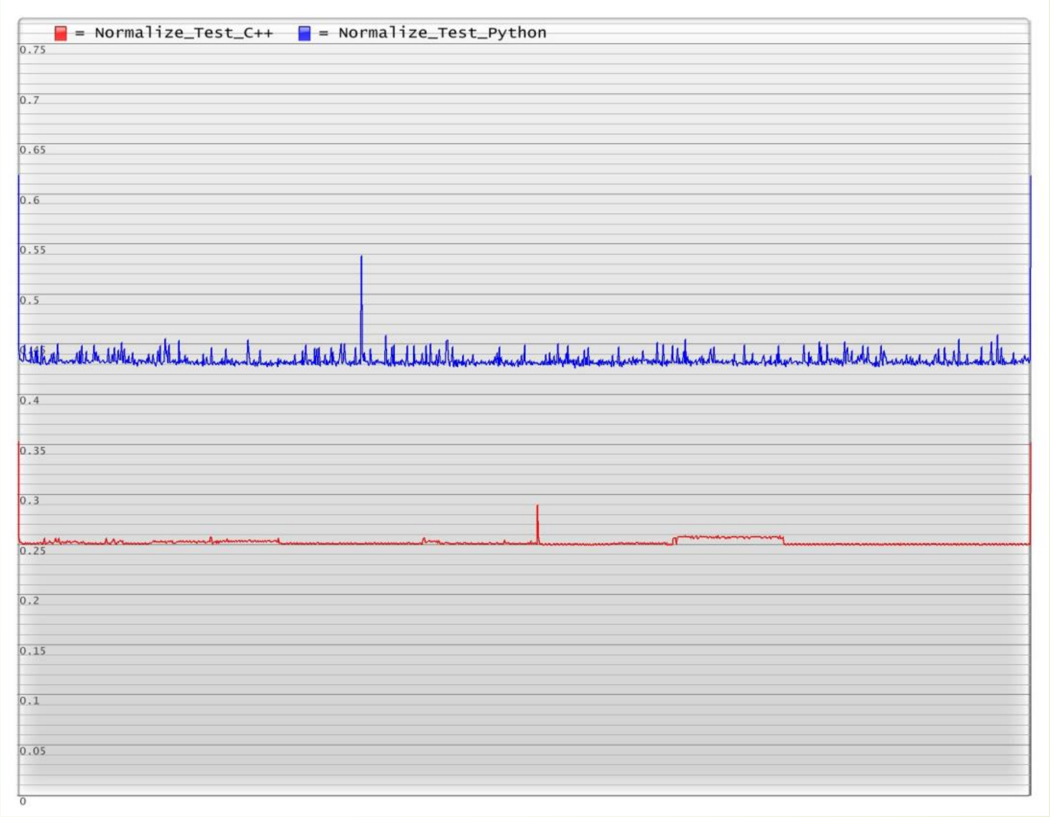
Runtime (ms)

C++ Python C++ Python

Results – The Bad

Vector Normalization



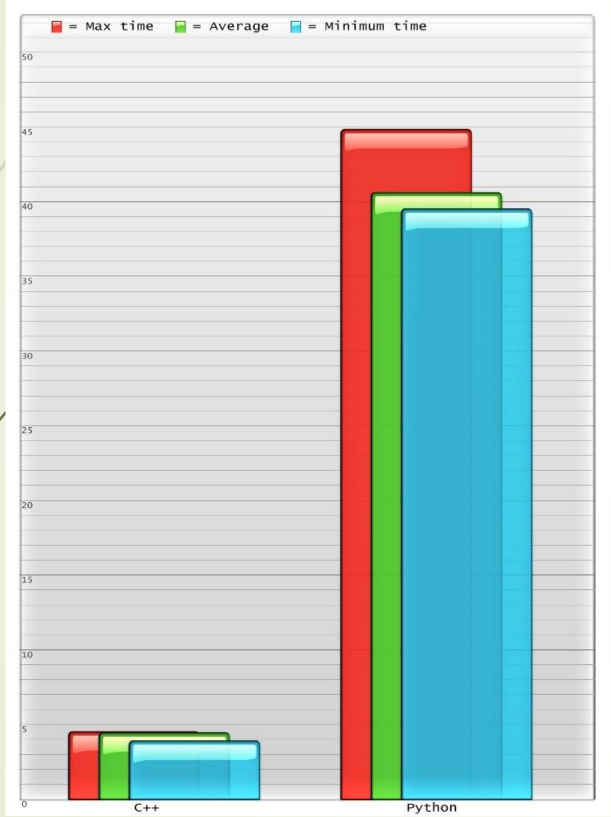


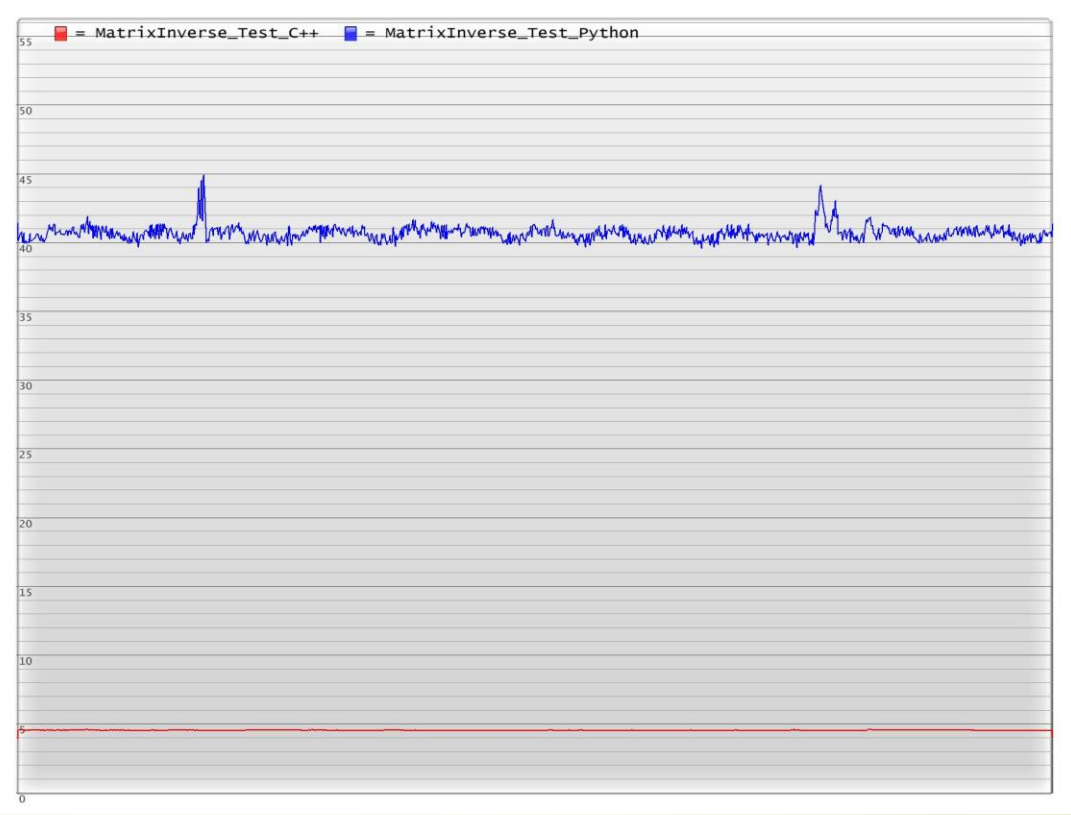
C++ Python

Jitter

Results – The Ugly

Matrix Inversion





C++ Python

Jitter

Conclusions

- Python ran an average of 4x slower than C++ in averaging all test results.
- Runtime jitter is more important for real-time applications than average execution times.
- Mathematical, memory intensive, or complex algorithms suffer the biggest performance impacts in Python.
- Utilizing Pythons built in methods or external modules can produce near or better than C++ performance.