A Brief Introduction to Python for those who know Java

- Developed by Guido van Rossum in the early 1990s
- Named after Monty Python
- Available on eniac
- Available for download from http://www.python.org

Advantages of Python

- Faster development: concise syntax, also no lengthy compile/linking steps
- Portable: programs run on major platforms without change
- Various built-in types: lists, dictionaries
- Large collection of support libraries: eg. NumPy for Matlab like programming

Recommended Reading

- On-line Python tutorials
  - The Python Tutorial (http://docs.python.org/tutorial/)
    - Dense but more complete overview of the most important parts of the language
    - See course homepage for other tutorials
- PEP 8- Style Guide for Python Code
  - http://www.python.org/dev/peps/pep-0008/
  - The official style guide to Python, contains many helpful programming tips

Which Python?

- Python 2.7
  - Current version on Eniac, so we’ll use it
  - Last stable release before version 3
- NOT Python 3
  - Many changes (including incompatible changes)
  - More existing third party software is compatible with Python 2 than Python 3 right now

Python interpreters

- On eniac
  - http://www.python.org/duke-
    - Python 2.7.2 (default, Aug 19 2011, 20:41:43) [GCC]
    - Type "help", "copyright", "credits" or "license" for more information.
    - GU - IDLE
A Silly Code Sample (in IDLE)

```python
x = 34 - 23  # A comment.
y = "Hello"   # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"  # String concat.
print x
print y
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`.
- For numbers `+` - `/` % are as expected.
- Special use of `+` for string concatenation.
- Special use of `%` for string formatting (as with printf in C)
- Logical operators are words (and, or, not) not symbols
- Simple printing can be done with `print`
- Indentation matters to the meaning of the code:
  - Block structure indicated by indentation
- The first assignment to a variable creates it.
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.

Whitespace is meaningful in Python

- Use a newline to end a line of code.
  - Use `\` when must go to next line prematurely.
- Block structure is indicated by indentation
  - The first line with less indentation is outside of the block.
  - The first line with more indentation starts a nested block
  - Often a colon appears at the start of a new block. (E.g. for function and class definitions.)

Dynamic typing – the key difference

- Java: *statically typed*
  - Variables are declared to refer to objects of a given type
  - Methods use type signatures to enforce contracts
- Python: *dynamic typed*
  - Variables come into existence when first assigned to
  - A variable can refer to an object of any type
  - All types are (almost) treated the same way
  - Main drawback: type errors are only caught at runtime

Assignment

- Binding a variable in Python means setting a name to hold a reference to some object.
  - Assignment creates references, not copies (like Java)
- A variable is created the first time it appears on the left side of an assignment expression: `x = 3`
- An object is deleted (by the garbage collector) once it becomes unreachable.
- Dynamic Typing: Names in Python do not have an intrinsic type.
  - Python determines the type of the reference automatically based on what data is assigned to it.
- BUT Strong Typing: Objects have fixed types.

Objects and types

- Objects everywhere: Every entity is an object!
- Strongly typed: Every object has an associated type, which it carries everywhere
- Built-in object types:
  - Number 10
  - String "hello", 'hello'
  - List `[1, 'abc', 44]`
  - Tuple `(4, 5)`
  - Dictionary `{'food': 'something you eat', 'lobster': 'an edible, underwater arthropod'}`
  - Files
- Missing: Arrays, Characters!
Sequence Types

1. Tuple
   - A simple immutable ordered sequence of items
     - Immutable: a tuple cannot be modified once created...
     - Items can be of mixed types, including collection types

2. Strings
   - Immutable
   - Very much like a tuple with different syntax
   - Regular strings use 8-bit characters. Unicode strings use 2-byte characters. (All this is changed in Python 3.)

3. List
   - Mutable ordered sequence of items of mixed types

Sequence Types 2

The three sequence types share much of the same syntax and functionality.
- Tuples
  >>> tu = (23, 'abc', 4.56, (2,3), 'def')
  >>> tu[1]     # Second item in the tuple.
  'abc'

- Strings
  >>> st = 'Hello World'; st = 'Hello World'
  >>> tu[1]     # Second item in the tuple.
  'abc'

Slicing: Return Copy of a Subset

>>> t = (23, 'abc', 4.56, (2,3), 'def')
Return a copy of the container starting at the first index, and stopping before the second index.
>>> t[1:4]
('abc', 4.56, (2,3))

You can also use negative indices when slicing.
>>> t[1:-1]
('abc', 4.56, (2,3))

Optional argument allows selection of every n\textsuperscript{th} item.
>>> t[1::2]
('abc', (2,3))

Operations on Lists Only 1

>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']

Operations on Lists Only 2

>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')  # index of first occurrence
1

>>> li.count('b')  # number of occurrences
2

>>> li.remove('b')  # remove first occurrence
>>> li
['a', 'c', 'b']
Operations on Lists Only 3

```python
>>> li = [5, 2, 6, 8]
>>> li.reverse() # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort()     # sort the list *in place*
>>> li
[2, 5, 6, 8]
>>> li.sort(some_function)
# sort in place using user-defined comparison
>>> sorted(li)    #return a copy sorted
```

Dictionaries: a mapping collection type

Creating, accessing & updating dictionaries

```python
>>> d = {'user': 'bozo', 'pswd': 1234}
>>> d['user']
'bozo'
>>> d['bozo']
Traceback (innermost last):
  File '<interactive input>', line 1, in ?
KeyError: bozo
>>> d = {'user': 'bozo', 'pswd': 1234}
>>> d['user'] = 'clown'
>>> d
{'user': 'clown', 'pswd': 1234}
```

* Assigning to an existing key replaces its value.
* Dictionaries are unordered & work by hashing

Useful Accessor Methods

```python
>>> d = {'user': 'bozo', 'p': 1234, 'l': 34}
>>> d.keys()    # List of current keys
['user', 'p', 'l']
>>> d.values()  # List of current values.
['bozo', 1234, 34]
>>> d.items()   # List of item tuples.
[('user', 'bozo'), ('p', 1234), ('l', 34)]
```

For Loops

```python
for <item> in <collection>:
    <statements>
```

* <item> can be more complex than a single variable name.
  ```python
  for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
      print x
  ```

* Range:
  ```python
  range(5) returns [0,1,2,3,4]
  ```
  So we can say:
  ```python
  for x in range(5):
      print x
  ```
  ```python
  xrange() returns an iterator that provides the same functionality more efficiently
  ```
List Comprehensions

```python
>>> li = [3, 6, 2, 7]
>>> [elem * 2 for elem in li]
[6, 12, 4, 14]
```

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

Filtered List Comprehension

```python
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

Filtered List Comprehension

```python
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produced.

Dictionaries as iterators return their set of keys

```python
>>> d = {'apple': 'red', 'banana': 'yellow', 'cherry': 'red'}
>>> for key in d:
...     d[key]
...     'yellow'
...     'red'
...     'red'
```

- A good way to get a list of the values in a dictionary:

```python
>>> [d[key] for key in d]
['yellow', 'red', 'red']
```

Useful things to know I

- Dictionaries as iterators return their set of keys

- A good way to get a list of the values in a dictionary:

```python
>>> [d[key] for key in d]
['yellow', 'red', 'red']
```

Useful things to know II

- Analogous to list comprehensions are `generator expressions`, written in `():`

- The class `defaultdict` in the `collections` module automatically initializes nonexistent dictionary values

```python
>>> from collections import defaultdict
>>> for val in (d[key] for key in d):
...     counters[val] += 1
>>> counters
defaultdict(<type 'int'>, {'red': 2, 'yellow': 1})
```

Computing a sorted list of n-grams

```python
from operator import itemgetter
def calc_ngram(inputstring, nlen):
    ngram_list = [inputstring[x:x+nlen] for x in xrange(len(inputstring)-nlen+1)]
    ngram_freq = () # dict for storing results
    for n in ngram_list:
        if n in ngram_freq:
            ngram_freq[n] += 1
        else:
            ngram_freq[n] = 1
    # human counting numbers start at 1
    # set reverse = False to change order of sort (ascending/descending)
    return sorted(ngram_freq.items(), \
                   key=itemgetter(1), reverse=True)
http://times.jayliew.com/2010/05/20/a-simple-n-gram-calculator-pyngram/
```
Functions in Python

(Functions later)

Defining Functions

- Function definition begins with `def` Function name and its arguments.
- No declaration of types of arguments or result

```python
def get_final_answer(filename):
    """Documentation String""
    line1
    line2
    return total_counter
...```

First line with less indentation is considered to be outside of the function definition.

```
'return' indicates the value to be sent back to the caller.
```

Function overloading? No.

- There is no function overloading in Python.
- Unlike Java, a Python function is specified by its name alone
  - The number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name.
  - Two different functions can’t have the same name, even if they have different numbers of arguments.

- But operator overloading – overloading `+`, `-`, etc. – is possible using special methods on various classes (see later slides)

Default Values for Arguments

- You can provide default values for a function’s arguments
- These arguments are optional when the function is called

```python
>>> def myfun(b, c=3, d="hello"):  
    return b + c

>>> myfun(5, 3, "hello")
>>> myfun(5, 3)
>>> myfun(5)
All of the above function calls return 8.
```

Keyword Arguments

- Functions can be called with arguments out of order
- These arguments are specified in the call
- Keyword arguments can be used for a final subset of the arguments.

```python
>>> def myfun(a, b, c):
    return a-b
>>> myfun(2, 1, 43)
1
>>> myfun(c=43, b=1, a=2)
1
>>> myfun(2, c=43, b=1)
1
```

Classes and Inheritance
Creating a class

class roster:
course = "cis521"
def __init__(self, name, dept):
    self.student_name = name
    self.student_dept = dept
def print_details(self):
    print "Name: " + self.student_name
    print "Dept: " + self.student_dept

student1 = roster("john", "cis")
student1.print_details()

Subclasses

- A class can extend the definition of another class
- Allows use (or extension) of methods and attributes already defined in the previous one.
- New class: subclass. Original: parent, ancestor or superclass

To define a subclass, put the name of the superclass in parentheses after the subclass's name on the first line of the definition.

```python
class ai_student(student):
    pass
```

- Python has no 'extends' keyword like Java.
- Multiple inheritance is supported.

Redefining Methods

- Very similar to over-riding methods in Java
- To redefine a method of the parent class, include a new definition using the same name in the subclass.
  - The old code won't get executed.
- To execute the method in the parent class in addition to new code for some method, explicitly call the parent's version of the method.
  ```python
  parentClass.methodName(self, a, b, c)
  ```
- The only time you ever explicitly pass self as an argument is when calling a method of an ancestor.
  - So: myOwnClass.methodName(a, b, c)

__init__ constructors in subclasses:

- UNLIKE Java:
- To execute the ancestor's __init__ method the ancestor's __init__ must be called explicitly
- The first line of the __init__ method of a subclass will often be:
  ```python
  parentClass.__init__(self, x, y)
  ```
  where parentClass is the name of the parent's class.

Special Built-In Methods and Attributes

```python
class student:
    ...
def __repr__(self):
    return "I'm named " + self.full_name + " - age: ", self.age
    ...

f = student("Bob Smith", 23)
print f
I'm named Bob Smith - age: 23
print f
"I'm named Bob Smith - age: 23"
```
Other Special Methods

- **Used to implement operator overloading**
  - Most operators trigger a special method, dependent on class
    - `__init__`: The constructor for the class.
    - `__len__`: Define how `len(obj)` works.
    - `__copy__`: Define how to copy a class.
    - `__cmp__`: Define how `==` works for class.
    - `__add__`: Define how `+` works for class
    - `__neg__`: Define how unary negation works for class

- Other built-in methods allow you to give a class the ability to use `[]` notation like an array or `()` notation like a function call.

---

Iterators

class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
    def __iter__(self):
        return self

>>> for char in Reverse('spam'):
    print char
m
a
p
s

---

Efficient memory usage

Eg: File Objects

```python
>>> for line in open("script.py"):
    ... print(line.upper())

IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 3)
```

Instead of

```python
>>> for line in open("script.py").readlines():
    ... print(line.upper())
```

---

Generators
Generators

- Defines an iterator with a function
- Maintains local state automatically

```python
def gensquares(N):
    for i in range(N):
        yield i ** 2
```

```python
>>> for i in gensquares(5):
    print(i)
0
1
4
9
16
```

Using generators

- Merging sequences:

```python
def merge(l, r):
    llen, rlen, i, j = len(l), len(r), 0, 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```

```python
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
    print g.next
1
2
3
4
5
```

The first time merge is called, it returns an iterator!!

Generators and exceptions

```python
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
    try:
        print g.next()
    except StopIteration:
        print 'Done'
        break
```

```
1
2
3
4
5
Done
```

List Generators

```python
>>> a = (x * x for x in xrange(5))
```

```python
>>> a
<generator object <genexpr> at 0x031A7A80>
```

```python
>>> for x in a:
    ...:     print x
    ...:
0
1
4
9
16
```

A directed graph class

```python
>>> d = DiGraph([(1,2), (1,3), (2,4), (3,1)])
>>> print d
1 -> 2
1 -> 3
2 -> 4
4 -> 3
4 -> 1
```
A directed graph class

```python
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
[1, 2, 4, 3]
>>> [v for v in d.search(1)]
[1, 2, 4, 3]
>>> [v for v in d.search(4)]
[4, 3, 1, 2]
>>> [v for v in d.search(2)]
[2, 4, 3, 1]
>>> [v for v in d.search(3)]
[3]
```

The `search` method returns a generator for the nodes that can be reached from a given node by following arrows “from tail to head”.

The constructor builds a dictionary (self.adj) mapping each node name to a list of node names that can be reached by following one edge (an “adjacency list”).

```python
class DiGraph:
    def __init__(self, edges):
        self.adj = {}
        for u,v in edges:
            if u not in self.adj:
                self.adj[u] = [v]
            else:
                self.adj[u].append(v)

def __str__(self):
    return '
'.join(['%s - %s' % (u,v) for u in self.adj for v in self.adj[u]])
```

```python
>>> d = DiGraph([(1,2),(1,3),(2,4),(4,3),(4,1)])
>>> d.adj
{1: [2, 3], 2: [4], 4: [3, 1]}
```

The `search` method

```python
class DiGraph:
    def search(self, u, visited=set()):
        # If we haven't already visited this node...
        if u not in visited:
            # yield it
            yield u
            # and remember we've visited it now.
            visited.add(u)
            # Then, if there are any adjacent nodes...
            if u in self.adj:
                # for each adjacent node...
                for v in self.adj[u]:
                    # search for all nodes reachable from *it*...
                    for w in self.search(v, visited):
                        # and yield each one.
                        yield w
```