Python Tutorial

INSTALLATION
Install Python on Windows

- Download Python 2.7.6 for Windows from http://www.python.org/ftp/python/2.7.6/python-2.7.6.msi
- Run the installer python-2.7.6.msi
- Follow the instructions to install Python
- The default installed directory should be like C:\Python27

It’s ok to run any Python 2.7.x. Just make sure you don’t install Python 3.x for this course.
Install Python on Mac

- Some Mac machines come with Python installed: try typing `python --version`
- If you need to install Python on your Mac: download Python 2.7.6 for Mac from [http://www.python.org/ftp/python/2.7.6/python-2.7.6-macosx10.6.dmg](http://www.python.org/ftp/python/2.7.6/python-2.7.6-macosx10.6.dmg) (for newer Macs on OSX 10.6 or higher)

It’s ok to run any Python 2.7.x. Just make sure you don’t install Python 3.x for this course.
Install Python on Linux

- If you are using Ubuntu, Python (probably version 2.6.x) is installed by default
- If you want to install Python 2.7.6: download the gzipped of bzipped source file from [http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.bz2](http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.bz2)
- Unzip the file, cd to the unzipped directory, and run the normal "/configure → make → sudo make install" commands

It’s ok to run any Python 2.7.x. Just make sure you don’t install Python 3.x for this course.
Python Tutorial

CRASH COURSE IN PYTHON
Python Interpreter

- Interactive interface to Python
  - On Windows: Start → All Programs → Python 2.6 → Python (command line)
  - On Linux: type `python`
  - Python prompt: `>>>`
  - To exit:
    - On Windows: Ctrl-Z + <Enter>
    - On Linux: Ctrl-D

- Run python program on Linux:
  ```bash
  % python filename.py
  ```
robert:~$ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>>
- **IDLE: the Python IDE**
  - On Windows: Start → All Programs → Python 2.6 → IDLE (Python GUI)
  - On Linux: type `idle`
 introduction to code actively, with Codecademy.

Hi there! People write programs to make computers do things. At the start, we can make your computer do some math for us (so we don't have to do it ourselves)!

Add any numbers you like. Why not try 3 + 4? Hit enter after you type them in (make sure to do this from now on after you complete the instructions).

>
For today’s exercises

1. Sign up for Codecademy now
2. Pick the Python course
3. Jump to an appropriate exercise sequence
Let’s Use LearnPython.org instead

- We are probably not going to use these slides ... because that’s boring, but try to use LearnPython instead. Anyways, these are backup slides.
The Basics

- Numbers: integers, long integers, floating points, and complex numbers

- Strings:
  - Single quotes: 'hello!'
  - Double quotes: "what's your name?"
  - Triple quotes: multi-line string
    ```
    '''This is multi-line string.
    '''
    ```

- Immutable: once created, cannot change

- Concatenation: 'hi ' 'there' same as 'hi ' + 'there'
- **Variables:** \( x = 1, \ y = 'NUS', \ x = y = z = 'SoC' \)

- **Identifier naming:**
  - 1\(^{st}\) char: a letter of the alphabet (upper or lowercase) or an underscore (‘_’)
  - The rest: letters (upper or lowercase), underscores (‘_’), or digits (0-9)
  - Case-sensitive: myname \(!=\) myName

- **Reserved words:**
  - and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while
- Strongly object oriented: everything is an object, including numbers, string, functions

- Statements and semicolons:
  - Don’t need to put ;) if you write one statement in one single line
    ```
    i = 5
    print i
    ```
  - Statements in one line are separated by (;)
    ```
    i = 5; print i
    ```
  - Continuing a line with (\)
    ```
    s = 'This is a string. \
    This continues the string.'
    ```
Indentation:

- No braces {} to mark blocks of code
- Leading whitespaces (spaces and tabs) are important
- Statements in the same block have same indentation

```python
i = 5
print i  # wrong

i = 5
print i  # correct
```

- Recommendation: consistently use a single tab or 2/4 spaces
Comments:

- Single line uses #
- Multi-line uses """"...""""

# this is single line comment
"""" this is multiple line comment """"
Operators and Expressions

- **Operators:**
  
  +, -, *, **, /, //, %,
  
  <<, >>, &, |, ^, ~,
  
  <, >, <=, >=, ==, !=,
  
  not, and, or

- **Expression:**

  length = 5
  breadth = 2
  area = length * breadth
  print 'Area is', area
  print 'Perimeter is', 2 * (length + breadth)

- **Assignment uses =, comparison uses ==**

- **Multiple assignments:**

  x, y = 2, 'abc'

Pretty print:

```
Try: print "He's", 5, 'years', 'old.'
```
Control Flow

- if...elif...else statement:

```python
number = 23
guess = int(raw_input('Enter an integer : '))
if guess == number:
    print 'Congratulations, you guessed it.'  # New block starts here
    print '(but you do not win any prizes!)'  # New block ends here
elif guess < number:
    print 'No, it is a little higher than that'  # Another block
    # You can do whatever you want in a block ...  
else:
    print 'No, it is a little lower than that'
    # you must have guess > number to reach here
print 'Done'
```
while statement:

number = 23
running = True
while running:
    guess = int(raw_input('Enter an integer : '))
    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False # this causes the while loop to stop
    elif guess < number:
        print 'No, it is a little higher than that.'
    else:
        print 'No, it is a little lower than that.'
else:
    print 'The while loop is over.'
    # Do anything else you want to do here
print 'Done'
for statement:

for i in range(1,5):
    print i
else:
    print 'the for loop is over'

- range(1,5) generates the list [1, 2, 3, 4]
- range(a, b, s) generates a list from a to b-1 with a step s
  - range(1,10,2) \rightarrow [1, 3, 5, 7, 9]
- **break statement:**
  - break from current loop
- **continue statement:**
  - skip the rest of the current iteration and continue to the next iteration
It’s Codecademy time!

- This is the Восток level of achievement programming contest as well!
- All teams that can complete the exercise should post a follow-up to the thread on Piazza.
It’s Codecademy time!

- This is the Восток level of achievement programming contest as well!
- All teams that can complete the exercise should post a follow-up to the thread on Piazza.
Functions

- Defining a function with `def`:

```python
def printMax(a, b):
    if a > b:
        print a, 'is maximum'
    else:
        print b, 'is maximum'
printMax(3, 4) # directly give literal values
x = 5
y = 7
printMax(x, y) # give variables as arguments
```
Local variables: variables declared inside a function

```python
>>> def func(x):
...     print 'x is', x
...     x = 2
...     print 'Changed local x to', x
...     
...     x = 50
>>> func(x)
x is 50
Changed local x to 2
>>> print 'x is still', x
x is still 50
```
Use `global` to explicitly assign a value to a variable declared outside the function

```python
>>> def func():
    ...    global x
    ...    print 'x is', x
    ...    x = 2
    ...    print 'Changed global x to', x
    ...
>>> x = 50
>>> func()
x is 50
Changed global x to 2
>>> print 'Value of x is', x
Value of x is 2
```
Default argument value: make some parameters optional by providing default values

```python
>>> def say(message, times = 1):
...     print message * times
... >>> say('Hello')
Hello
>>> say('World', 5)
WorldWorldWorldWorldWorld
```

Note: only those at the end of the parameter list can be given default value

- `def func(a, b=5) is valid`,
- `def func(a=5, b) is not valid`
Keyword arguments: use names instead of positions to specify the arguments to the function

```python
>>> def func(a, b=5, c=10):
...     print 'a is', a, 'and b is', b, 'and c is', c
...     ...

>>> func(3, 7)
a is 3 and b is 7 and c is 10
>>> func(25, c=24)
a is 25 and b is 5 and c is 24
>>> func(c=50, a=100)
a is 100 and b is 5 and c is 50
```
- **Use** `return` **to break out of a func and/or return a value**

```python
def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print maximum(2, 3)
```
Modules

- Module: a file containing all functions and variables that you have defined
- The module file should end with .py
- Use `import` to import a module:
  - Like Java `import` and C++ `include`
  - 3 formats:
    - `import somefile`
    - `from somefile import *`
    - `from somefile import className`
- Import standard library and math library:
  - `import sys`
  - `import math`
# Make your own module:

```python
# Filename: mymodule.py
def sayhi():
    print 'Hi, this is mymodule speaking.'

version = '0.1'
# End of mymodule.py

import mymodule
mymodule.sayhi()
print 'Version', mymodule.version
```

Output:
Hi, this is mymodule speaking.
Version 0.1

Save the module into mymodule.py

Import mymodule from the same dir
Data Structures

- Built-in data structures: list, tuple, dictionary

- List:
  - Specified by \([\text{item0}, \text{item1}, ...]\)
  - Stores a sequence of items
  - List is mutable: we can add, remove, change items

- List can store different types of items:
  - \([1, 'nus', [3, 'soc'], \text{None}, \text{True}]\)
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']
print 'I have', len(shoplist), 'items to purchase.'
print 'These items are:', # Notice the comma at end of the line
for item in shoplist:
    print item,
print '\nI also have to buy rice.'
shoplist.append('rice')
print 'My shopping list is now', shoplist
print 'I will sort my list now'
shoplist.sort()
print 'Sorted shopping list is', shoplist
print 'The first item I will buy is', shoplist[0]
olditem = shoplist[0]
del shoplist[0]
print 'I bought the', olditem
print 'My shopping list is now', shoplist
- **Tuple**
  - Specified by `(item0, item1, ...)`
  - Like lists except they are immutable: cannot be modified
  - Used when you can assume the collection of items will not change
- **Tuple can store different types of items:**
  - `(1, 'nus', [3, 'soc'], None, True, (1, 'a'))`
  - Empty tuple: `()`
  - Tuple with one item: `(1, )` not `(1)`
zoo = ('wolf', 'elephant', 'penguin')
print 'Number of animals in the zoo is', len(zoo)
new_zoo = ('monkey', 'dolphin', zoo)
print 'Number of animals in the new zoo is', len(new_zoo)
print 'All animals in new zoo are', new_zoo
print 'Animals brought from old zoo are', new_zoo[2]
print 'Last animal brought from old zoo is', new_zoo[2][2]

Output:
Number of animals in the zoo is 3
Number of animals in the new zoo is 3
All animals in new zoo are ('monkey', 'dolphin', ('wolf', 'elephant', 'penguin'))
Animals brought from old zoo are ('wolf', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Tuples and the `print` statement: one of the most common usage of tuple, use `%` to format output

```python
age = 22
name = 'Alex'
print '%s is %d years old' % (name, age)
print 'Why is %s playing with that python?' % name
```

Output:
Alex is 22 years old
Why is Alex playing with that python?
- Dictionary:
  - hash with key/value pairs
  - Keys must be unique
  - Specified by:
    - `{key0:value0, key1:value1, ...}`
ab = { 'Alex' : 'alex@gmail.com',
    'Bob' : 'bob@yahoo.com'}
print "Alex's email is %s" % ab['Alex']
# Adding a key/value pair
ab['Cindy'] = 'cindy@gmail.com'
# Deleting a key/value pair
del ab['Alex']
print '
There are %d contacts in the address-book
' % len(ab)
for name, address in ab.items():
    print 'Contact %s at %s' % (name, address)
if 'Cindy' in ab:
    print "\nCindy's email is %s" % ab['Cindy']

Output:
Alex's email is alex@gmail.com
There are 2 contacts in the address-book
Contact Bob at bob@yahoo.com
Contact Cindy at cindy@gmail.com
Cindy's email is cindy@gmail.com
- **Sequences:**
  - Examples: list, tuple, string

- **2 main features of sequences:**
  - Indexing: fetch a particular item
    - `[1,'a'][1]`, `(1,'a')[1]`, `'hello'[1]
  - Slicing: retrieve a slice of the sequence
    - `'hello'[1:4] => 'ell`

- **Key difference:**
  - Tuples and strings are immutable
  - Lists are mutable
Indexing and slicing a sequence:

```
list[0]  list[1]  ...
    ↓       ↓       ↓
list[-2] list[-1] ...
    ↓       ↓       ↓
list = ['apple', 'mango', 'carrot', 'banana']
          ↓
list[1:3]

>>> a = "Singapore"
>>> a[2]
'n'
>>> a[:]
'Singapore'
>>> a[2:5]
'nga'
```
Sequence operators:

- **in**: boolean test whether an item is inside a sequence
  
  ```python
  1 in [2, 'a', 1]  \rightarrow  True
  'a' in 'abcd'  \rightarrow  True
  ```

- **+**: produces a new sequence by joining two
  
  ```python
  (1, 2) + (3, 4)  \rightarrow  (1, 2, 3, 4)
  'ab' + 'cd'  \rightarrow  'abcd'
  ```

- **\***: produces a new sequence by repeating itself
  
  ```python
  [1, 2] \ *\ 2  \rightarrow  [1, 2, 1, 2]
  'Hello' \ *\ 3  \rightarrow  'HelloHelloHello'
  ```
- **Sequence methods:**
  - `len(s)`: return length of the sequence `s`
  - `min(s)` and `max(s)`: return the min and max value in `s`
  - `list(s)`: convert a sequence to a list
List: \texttt{+} vs \texttt{extend()} vs \texttt{append()}

- \texttt{+} creates a fresh list (new memory reference)
- \texttt{extend} a list with another list
- \texttt{append} a list with another item

```python
>>> a = [1, 2]
>>> b = [3, 4]
>>> a + b
[1, 2, 3, 4]
>>> a + b
[1, 2, 3, 4]
>>> a.append([5, 6])
>>> a
[1, 2, [5, 6]]
>>> a.append(7)
>>> a
[1, 2, [5, 6], 7]
>>> b.extend([5, 6])
>>> b
[3, 4, 5, 6]
```
More list methods:

- \( s \text{.count}(x) \): counts the occurrences of an element in a list
- \( s \text{.index}(x) \): finds the first location of an element in a list
- \( s \text{.remove}(x) \): searches for and removes an element in a list
- \( s \text{.sort}() \): sorts a list
- \( s \text{.reverse}() \): reverses the order of a list
- References: when you bind a variable and an object, the variable only refers to the object and does not represent the object itself
- A subtle effect to take note:

```python
list1 = ['a', 'b', 'c']
list2 = list1  # list2 points to the same list object
list3 = list1[:]  # list3 points to a new copy
```
- More string methods
  - `str1.startswith(str2)`: check whether `str1` starts with `str2`
    
    `'Hello'.startswith('He')` → `True`
  - `str2 in str1`: check whether `str1` contains `str2`
    
    `'ell' in 'Hello'` → `True`
  - `str1.find(str2)`: get the position of `str2` in `str1`; -1 if not found
    
    `'Hello'.find('ell')` → `1`
String $\leftrightarrow$ list

- `delimiter.join(list)`: join the items in list with delimiter
  
  `'_' . join(['a', 'b', 'c']) $\rightarrow$ 'a_b_c'

- `str.split(delimiter)`: split the str with delimiter into a list
  
  `'a_b_c'.split('_') $\rightarrow$ ['a', 'b', 'c']
More dictionary methods:

- `a[k] = x`: sets a value in the dictionary
- `a.has_key(k)`: tests for the presence of a keyword
- `a.get(k, d)`: returns a default if a key is not found
- `a.keys()`: returns a list of keys from a dictionary
- `a.values()`: returns a list of values
Typing in Python

- Built-in types: `str`, `bytes`, `list`, `tuple`, `set`, `dict`, `int`, `float`, `complex`, `bool`  

- Dynamic typing: determines the data types of variable bindings automatically

  ```python
  var = 2
  var = 'hello'
  ```

- Strong typing: enforces the types of objects

  ```python
  >>> print 'The answer is ' + 23
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: cannot concatenate 'str' and 'int' objects
  >>> print 'The answer is ' + str(23)
  The answer is 23
  ```
It’s **codecademy** time!

- This is the **Project Gemini** level of achievement programming contest as well!
- All teams that can complete the exercise should post a follow-up to the thread on Piazza.
Object-oriented Programming

- Class: a data type
- Object: an instance of the class
- Fields: variables that belong to an object or class
  - Two types: instance variables and class variables
- Methods: functions that belong to a class
- Fields and methods are referred to as the attributes of that class
The **self**:  
- The first parameter of a class method is the `self` (similar to `self` in C++ and `this` in Java)  
- But you don’t need to specify `self` when calling the method
Creating a simple class:

class Person:
    def sayHi(self):
        print 'Hi there!'

p = Person()
print p
p.sayHi()

Output:
<__main__.Person instance at 0xf6fcb18c>
Hi there!
The `__init__` method:
- Is run as soon as an object is instantiated
- Analogous to a constructor in C++ and Java

```python
class Person:
    def __init__(self, name):
        self.name = name
    def sayHi(self):
        print 'Hello, my name is', self.name

p = Person('Jack')
p.sayHi()
```

Output:
Hello, my name is Jack
Class and object variables:

- Class variable: accessed by all objects of the class
  - Changes will be seen by all objects
- Object variables: owned by each individual object of the class
  - Not shared by other objects of the same class
class Person:
    '''Represents a person.''
    population = 0
    def __init__(self, name):
        '''Initializes the person's data.''
        self.name = name
        print '(Initializing %s)' % self.name
        # When this person is created, he/she
        # adds to the population
        Person.population += 1
    def __del__(self):
        '''I am dying.''
        print '%s says bye.' % self.name
        Person.population -= 1
        if Person.population == 0:
            print 'I am the last one.'
        else:
            print 'There are still %d people left.' % Person.population
    def sayHi(self):
        '''Greeting by the person.''
        print 'Hi, my name is %s.' % self.name
    def howMany(self):
        '''Prints the current population.''
        if Person.population == 1:
            print 'I am the only person here.'
        else:
            print 'We have %d persons here.' % Person.population
alex = Person('Alex')
alex.sayHi()
alex.howMany()
bob = Person('Bob')
bob.sayHi()
bob.howMany()
alex.sayHi()
alex.howMany()

Output:
(Initializing Alex)
Hi, my name is Alex.
I am the only person here.
(Initializing Bob)
Hi, my name is Bob.
We have 2 persons here.
Hi, my name is Alex.
We have 2 persons here.
Bob says bye.
There are still 1 people left.
Alex says bye.
I am the last one.
Inheritance:

- Implement a type and subtype relationship between classes
- Reuse of code
- Multiple inheritance
- Declared by:

  ```
  class DerivedClass(Base1, Base2, ...)
  ```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print '(Initialized SchoolMember: %s)' % self.name
    def tell(self):
        '''Tell my details.'''
        print 'Name:"%s" Age:"%s"' % (self.name, self.age),

class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print '(Initialized Teacher: %s)' % self.name
    def tell(self):
        SchoolMember.tell(self)
        print 'Salary: "%d"' % self.salary

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print '(Initialized Student: %s)' % self.name
    def tell(self):
        SchoolMember.tell(self)
        print 'Marks: "%d"' % self.marks
I/O

- Files:
  - Create an object of the `file` class to use the `read`, `readline`, or `write` method

- Write to a file:

  ```python
  f = file('file.txt', 'w')
  f.write(str)
  f.close()
  ```

- Read from a file:

  ```python
  f = file('file.txt')
  for line in f:
    print line
  f.close()
  ```

- ‘w’ for write
- ‘r’ for read
- ‘a’ for append
It’s Codecademy time!

- Try out the last exercise

File Input/Output – The Devil’s in the Details
(Exercises 5-)
- **Pickle:**
  - Use the `pickle` module to store any object to a file so that you can get it back later intact → storing object persistently
  - Another module `cPickle` is written in C, and is up to 1000 times faster
import cPickle as p
# import pickle as p

shoplistfile = 'shoplist.data'
shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = file(shoplistfile, 'w')
p.dump(shoplist, f) # dump the object to a file
f.close()
del shoplist # remove the shoplist

# Read back from the storage
f = file(shoplistfile)
storedlist = p.load(f)
print storedlist
Exceptions

- Errors are objects
  - More specific kinds of errors are subclasses of the general Error class

- Catch errors:
  
  ```python
  try ... except ...
  try ... except ... else ...
  try ... except ... else ... finally ...
  ```

- Raise errors:
  
  ```python
  raise ...
  ```
while True:
    try:
        x = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "That was no valid number. Try again..."
Standard Library

- The `sys` module:
  - Contains system specific functionality
  - **Use:** `import sys`
  - `sys.argv`: list of arguments
  - `sys.exit()`
  - `sys.version, sys.version_info`: Python version information
  - `sys.stdin, sys.stdout, sys.stderr`
  - ...
The `os` module:

- Generic operating system functionality
- Important if you want to make your program platform-independent
- **Use:** `import os`
- `os.sep`: windows → `\\`, linux → `/`
- `os.name`: windows → `nt`, linux → `posix`
- `os.getcwd()`: get current working directory
- `os.getenv()`, `os.putenv()`: get and set environment variables
- `os.listdir()`: get names of all files in the specified directory
- `os.remove()`: delete a file
- `os.system()`: run a shell command
- `os.linesep`: windows → `
`, linux → `
`, mac → ``
- `os.path.split()`, `os.path.isfile()`, `os.path.isdir()`,...
More Python

- Special methods:
  - Used to mimic certain behavior
    - E.g.: to use indexing `x[key]` for your class, you implement the `__getitem__()` method
  - `__init__(self, ...):` called to instantiate an object
  - `__del__(self):` called just before the object is destroyed
  - `__str__(self):` called when we print the object or use `str()`
  - `__lt__(self, other):` called when less than `<` is used
  - `__getitem__(self, key):` called when `x[key]` is used
  - `__len__(self):` called when `len()` is used
Random numbers:

- Print a random number in [0,1):
  
  ```python
  import random
  print random.random()
  ```

- `randrange(a, b)`: chooses an integer in the range [a, b)

- `uniform(a, b)`: chooses a floating point number in the range [a, b)

- `normalvariate(mean, sdev)`: samples the normal (Gaussian) distribution
List comprehension:

- Derive a new list from existing lists
- Similar to the list comprehension in Haskell
- Python programmers use list comprehension extensively

Syntax: `[expression for name in list]`

```python
>>> [2*i for i in [2,3,4]]
[4, 6, 8]
>>> [n * 3 for (x, n) in [('a', 1), ('b', 2), ('c', 3)]
[3, 6, 9]
>>>```
Filtered list comprehension:

- Use filter condition
- \([\text{expression } \text{for } \text{name in list if } \text{filter}]\)

>>> \([2\times i \text{ for } i \text{ in } [2,3,4] \text{ if } i > 2]\)
\([6, 8]\)

More examples:

>>> \([x\times y \text{ for } x \text{ in } [1,2,3,4] \text{ for } y \text{ in } [3,5,7,9]]\)
\([3, 5, 7, 9, 6, 10, 14, 18, 9, 15, 21, 27, 12, 20, 28, 36]\)

>>> \([(x,y) \text{ for } x \text{ in } [1,3,5] \text{ for } y \text{ in } [2,4,6] \text{ if } x < y]\)
\([(1, 2), (1, 4), (1, 6), (3, 4), (3, 6), (5, 6)]\)

>>> \([n \times 2 \text{ for } n \text{ in } [m + 1 \text{ for } m \text{ in } [3,2,4]]]\)
\([8, 6, 10]\)
It’s Codecademy time – Last one

- Try out an advanced exercise

- Advanced Topics in Python
  - Iteration Nation (Exercises 1-3)
  - List Comprehensions (Exercises 4-6)
  - List Slicing (Exercises 7-11)
Aggregating function arguments:

- You can use * or ** to aggregate arguments into a tuple or dictionary

```python
def fun(a, *args):
    print a
    print args

fun(1, 3, 'a', True)

Output:
1
(3, 'a', True)
```

```python
def fun(a, **args):
    print a
    print args

fun(1, b=3, c='a', d=True)

Output:
1
{'c': 'a', 'b': 3, 'd': True}
```
Lambda forms:

- Create anonymous functions at runtime
- Powerful when used with `filter()`, `map()`, `reduce()`

```python
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>> print filter(lambda x: x % 3 == 0, foo)
[18, 9, 24, 12, 27]
>>> print map(lambda x: x * 2 + 10, foo)
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>> print reduce(lambda x, y: x + y, foo)
139
```
The `exec` and `eval` statements:

- **exec**: execute Python statements stored in a string or file
- **eval**: evaluate valid Python expressions stored in a string

```python
>>> exec('a = "Hi " + "there!"; print a')
Hi there!
>>> eval('2**3')
8
```
The `repr()` and backticks (``) statement:
- Return a printable representation of the object

```python
>>> list = [1, 'a', True, [33]]
>>> repr(list)
"[1, 'a', True, [33]]"
>>> `list`
"[1, 'a', True, [33]]"
```
OTHER EXERCISES
Practice 1 - Python

- Write a program to count word occurrences in a file.
  - Convert all words to lowercase
  - Excludes numbers and punctuation
  - Print the words and word counts by descending frequency
  - Reads the file name as the only argument
    
    % python count_words.py filename.txt
Practice 2 - Python

- The “paper, scissors, stone” game: write a program to play “paper, scissors, stone” with the computer
  - User chooses how many points are required for a win
  - User keys in one of the three selections: (p)aper, (s)cissors, or s(t)one
  - Computer randomly generates one selection
Welcome to Paper, Scissors, Stone!

How many points are required for a win? 3

Choose (p)aper, (s)cissors, or s(t)one? t
Human: stone  Computer: paper  Computer wins!
Score: Human 0  Computer 1

Choose (p)aper, (s)cissors, or s(t)one? t
Human: stone  Computer: scissors  Human wins!
Score: Human 1  Computer 1

Choose (p)aper, (s)cissors, or s(t)one? p
Human: paper  Computer: paper  A draw
Score: Human 1  Computer 1

Choose (p)aper, (s)cissors, or s(t)one? s
Human: scissors  Computer: paper  Human wins!
Score: Human 2  Computer 1

Choose (p)aper, (s)cissors, or s(t)one? t
Human: stone  Computer: scissors  Human wins!
Score: Human 3  Computer 1

Final Score: Human 3  Computer 1