Introduction To Python

Technical Issues

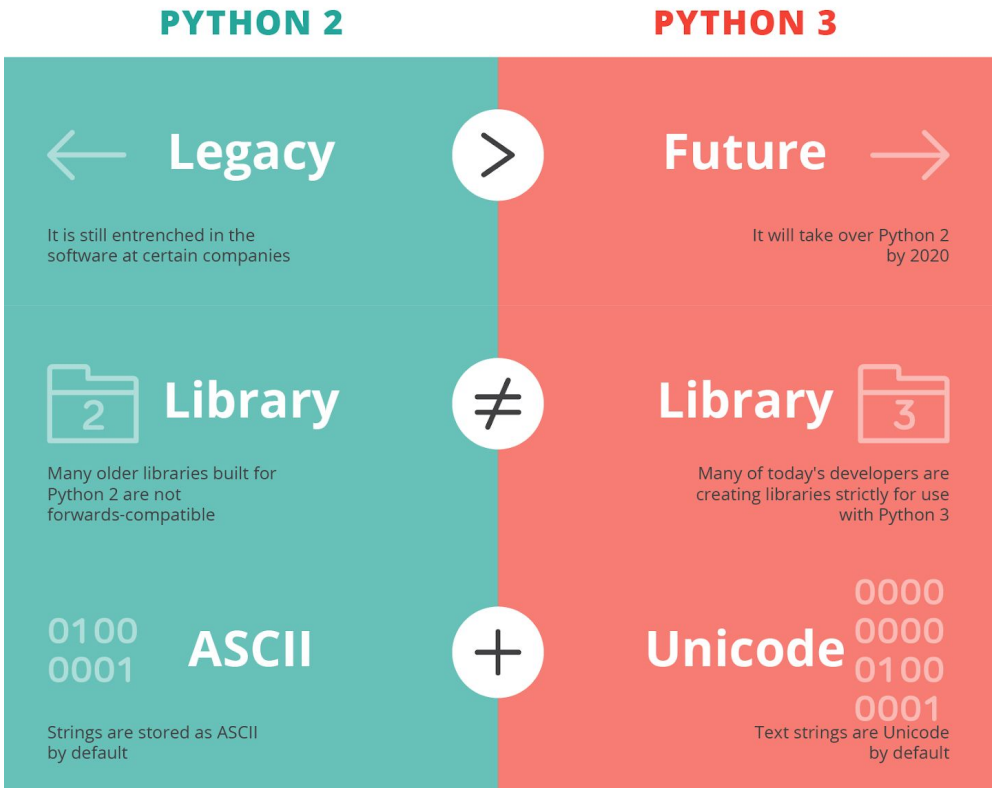
Installing & Running Python

Python

- Open source general-purpose language
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortan
- Easy-ish to interface with C++ (via SWIG)

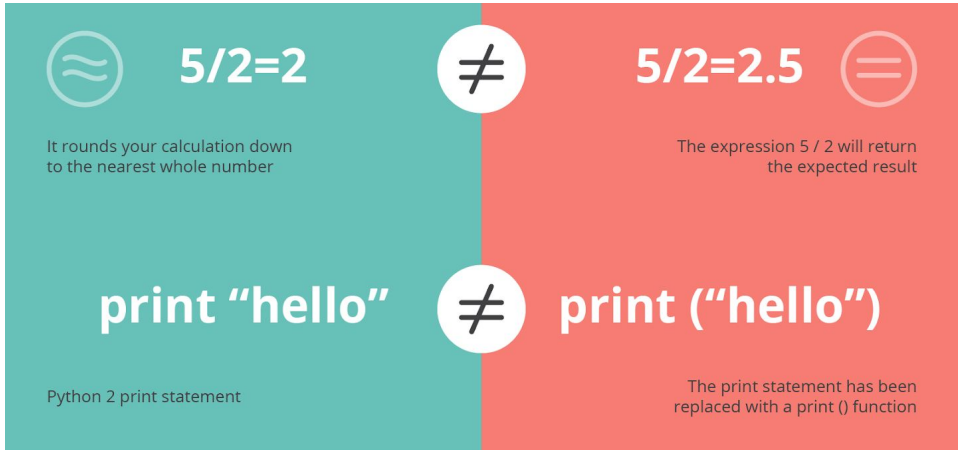
Version..?

- 2.7.x
- 3.7.x
- We are going to use python3 for most time.



Version..?

- 2.7.x
- 3.7.x
- We are going to use python3 for most time.



Installing & Running Python

- Python comes pre-installed with Mac OSX and Linux.
- Windows binaries from http://python.org/

In this course …

- We are going to use Anaconda package.
  - Anaconda package includes: Python's interpreter, notebook-style environment, many Python extension packages …

First, Run some setup code for this notebook. You don't have to edit these.

[1]:
```
# import libraries
import random
import numpy as np
import matplotlib.pyplot as plt
from cs231n.data_utils import load_CIFAR10

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2
```

The Python Interpreter

- Interactive interface to Python

```
PS C:\Users\steven-lee\Data Programming Lab> python
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Python interpreter evaluates inputs:

```
>>> 3*(7+2)
27
```

- To exit Python:

```
>>> exit()
PS C:\Users\steven-lee\Data Programming Lab>
```

Batteries Included

- Large collection of proven modules included in the standard distribution.
  - https://docs.python.org/3/py-modindex.html
- Additional Python Packages:
  - https://pypi.org/

numpy

- Offers Matlab-ish capabilities within Python
- Fast array operations
- 2D, multi dimensional arrays, linear algebra …

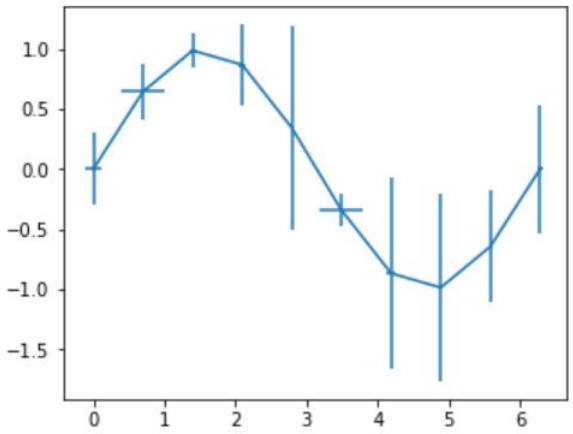
matplotlib

- High quality plotting library.

4. Plot with Error Bars

```
x = np.linspace(0, 2*np.pi, 10)
y = np.sin(x)
xerr = npr.rand(10)/3
yerr = npr.rand(10)
plt.errorbar(x, y, yerr, xerr)
plt.show()
```

[7]:
```
# Practice here!
```



The Basics

A Code Sample

- High quality plotting library.

```
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)
```

```
12
Hello World
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`.
- For numbers,
  - `+` Sum, *tuple, list, or string concatenation
  - `-` Subtract
  - `*` Multiplication
  - `**` Power
  - `/` Divide (float default, for Python 3)
  - `//` Quotient (not a comment line!)
  - `%` Remainder, *string formatting (as with printf in C)

Enough to Understand the Code

- Logical operators are words, not symbols: `and, or, not`.
- We print with print function (not a statement):

```
print("Hi")
```

- 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.

Basic Datatypes

- Integers, floats

```
x = 5 // 2
z = 5 / 2
print("5 // 2\t= ", x, "\t, type is ", type(x))
print("5 / 2\t= ", z, "\t, type is ", type(z))
```

```
5 // 2  =  2    , type is  <class 'int'>
5 / 2   =  2.5  , type is  <class 'float'>
```

Basic Datatypes

- Bool

```
x = (1 > 2)
y = (1 != 2)
print("1 > 2         is", x, ", type is", type(x))
print("1 != 2        is", y, " , type is", type(y))
print("x or y        is", x or y)
print("x and not y   is", x and not y)
```

```
1 > 2         is False , type is <class 'bool'>
1 != 2        is True  , type is <class 'bool'>
x or y        is True
x and not y   is False
```

Basic Datatypes

- Strings
  - You can use “” or ‘’ to specify: `'abc'  "abc"`
  - Unmatched can occur within the string: `"matt's"`
  - Use tripe double-quotes for multi-line strings of strings that contain both ‘ and “ inside of them: `"""a'b"c"""`

Type Conversion

- String to integer: + is numerical sum here.

```
20 + int("5")
```

```
25
```

- Integer to string: + is concatenation here.

```
str(20) + "5"
```

```
'205'
```

Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines
  - Use a newline to end a line of code. (use \ when must go next line prematurely.)
  - No braces to mark blocks of code in Python … use consistent indentation instead.

Comments

- Start comments with # - the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Assignment

- Binding a variable in Python means setting a name to hold a reference to some object.
  - Assignment creates references, not copies.
- Names in Python do not have an intrinsic type. Objects have types.
  - Python determines the type of the reference automatically based on the data object assigned to it.

Assignment

- You create a name the first time it appears on the left side of an assignment expression: `x = 3`
- Multiple Assignment: You can also assign to multiple names at the same time: `x, y = 2, 3`
- A reference is deleted via garbage collection after any names bound to it have passed out of scope.

Accessing Non-Existent Names

- If you try to access a name before it's been created, you'll get an error.

```
a
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-3f786850e387> in <module>
----> 1 a

NameError: name 'a' is not defined
```

Naming Rules

- Names are case sensitive and cannot start with a number.
- They can contain letters, numbers, and underscores.
  - `Bob bob _bob _2_bob bob_2 BoB`
- There are some 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`

Reference Semantics in Python

Understanding Reference Semantics

- Assignment manipulates references.
  - `x = y` does not make a copy of the object `y` references
  - `x = y` makes `x` reference the object `y` references.

Understanding Reference Semantics

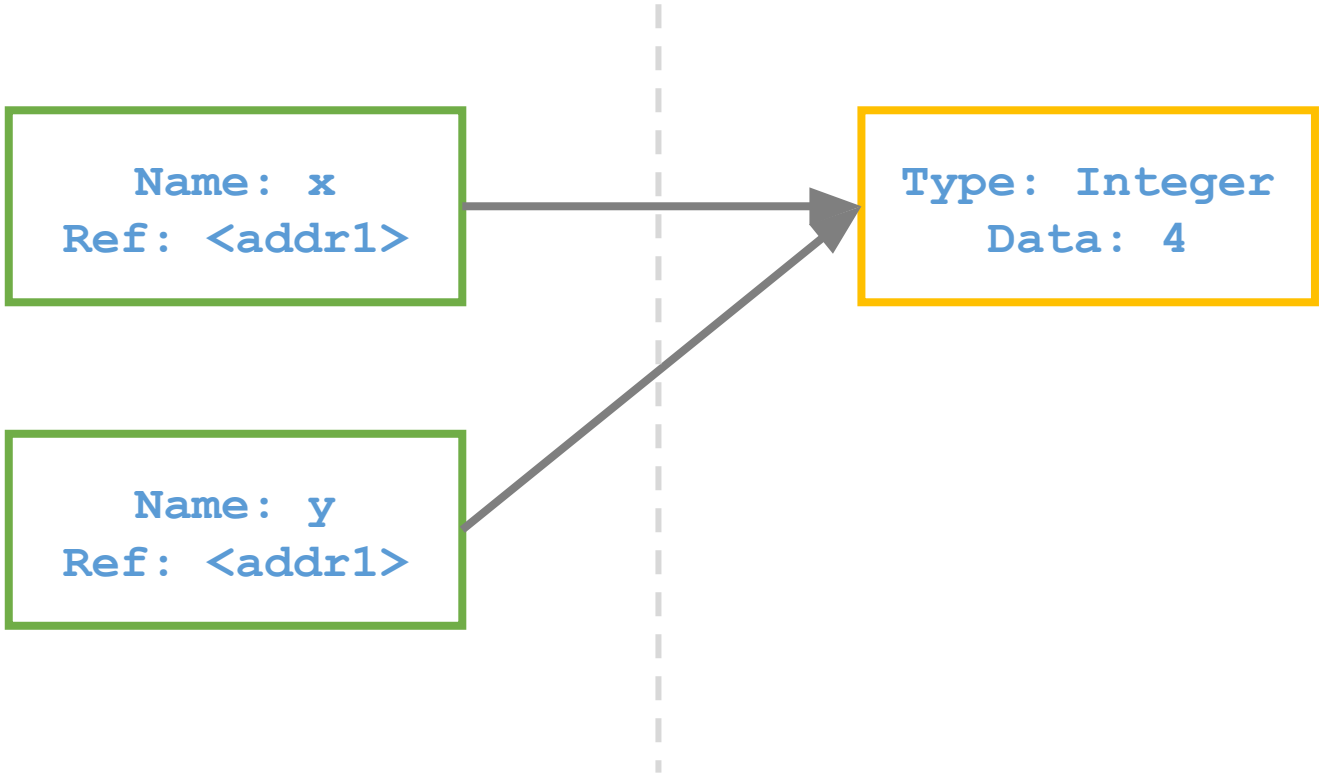
- `is` vs `==`
  - `is` returns true if two variables reference to the same object.
  - `==` returns true if the objects referred to by the variables are equal.

Understanding Reference Semantics

- Immutable Data Types: integers, floats, strings, bool, tuple
  - `x = 4`

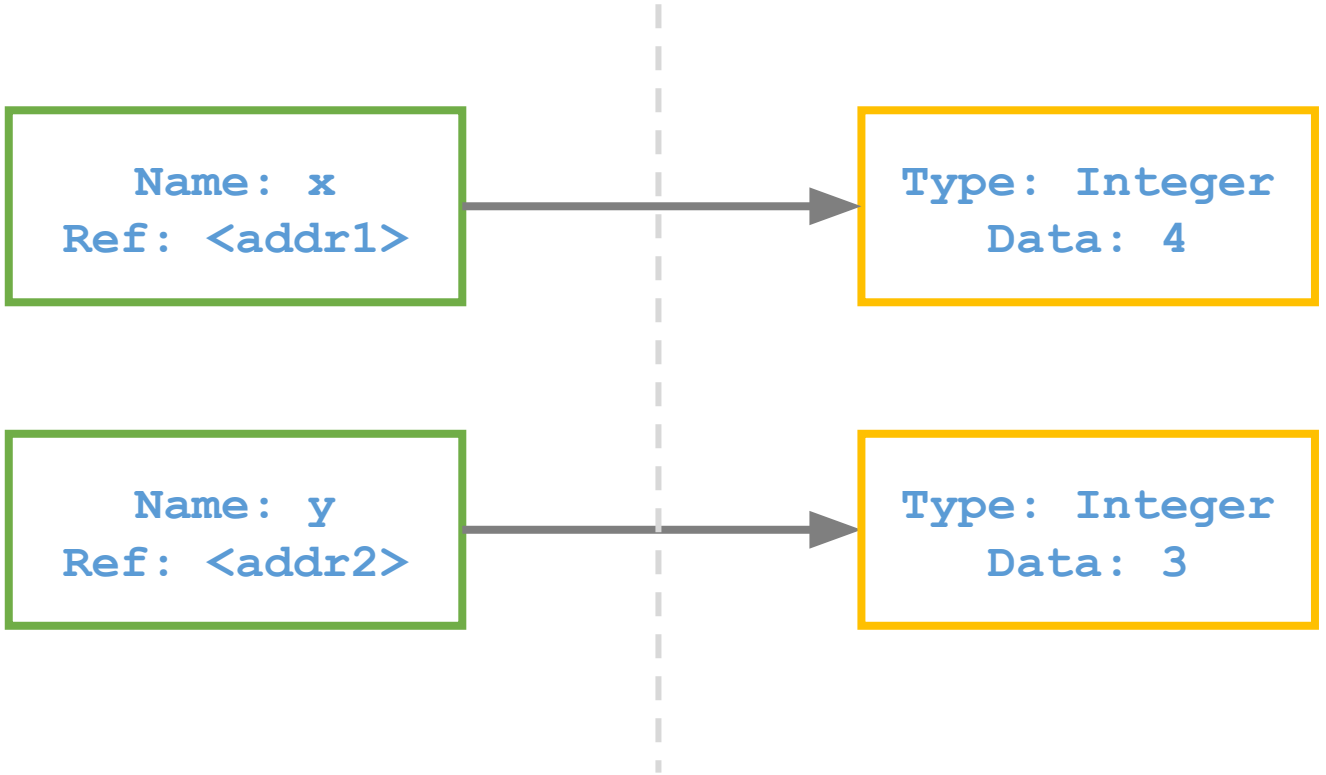
Understanding Reference Semantics

- Immutable Data Types: integers, floats, strings, bool, tuple
  - `x = 4`
  - `y = x`



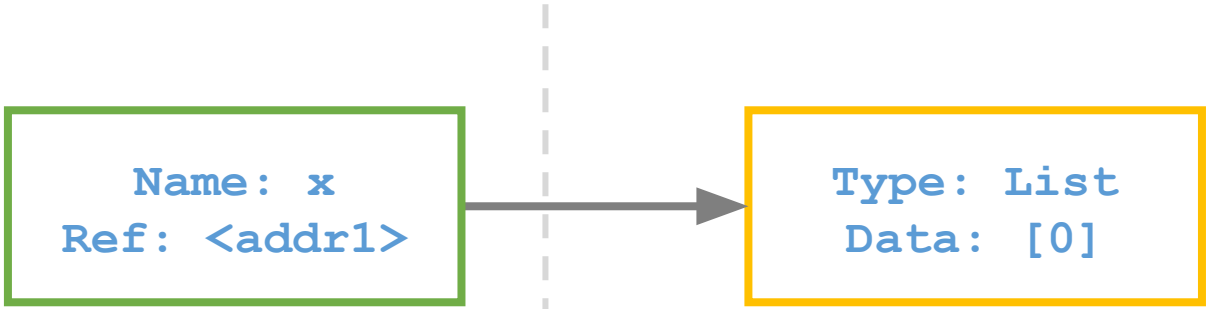
Understanding Reference Semantics

- Immutable Data Types: integers, floats, strings, bool, tuple
  - `x = 4`
  - `y = x`
  - `y = 3`



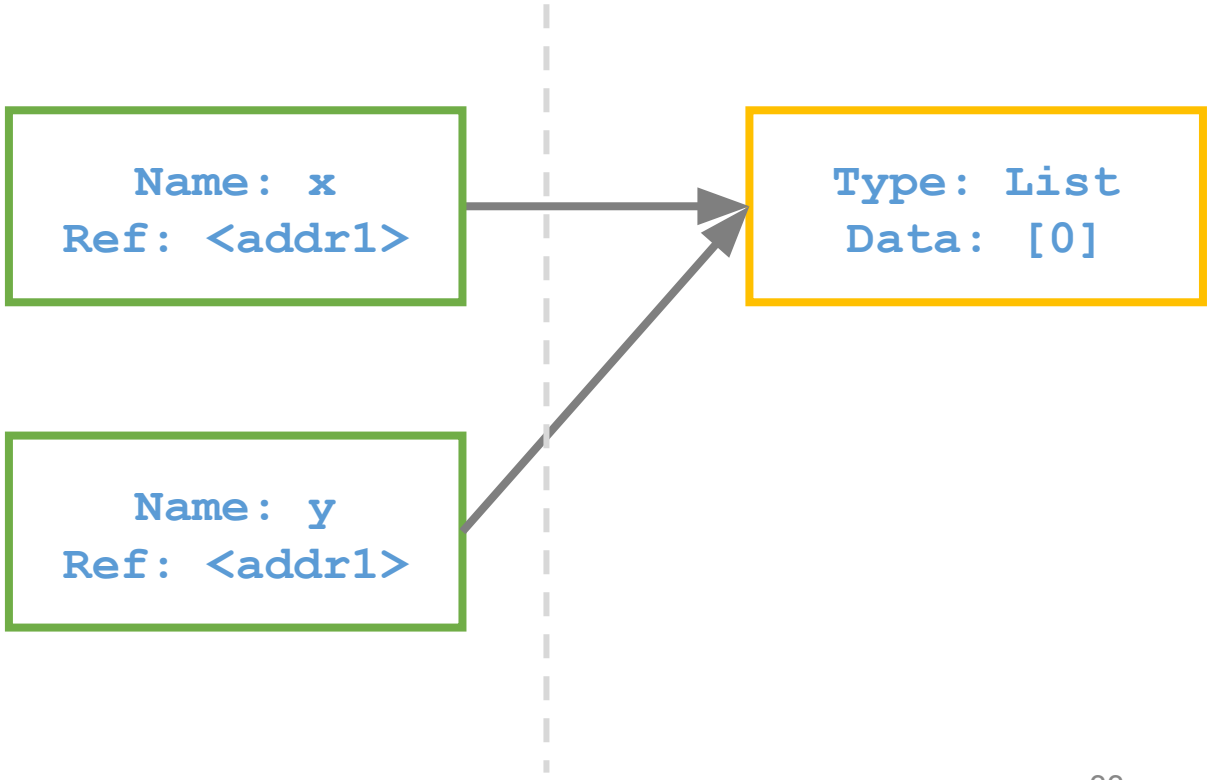
Understanding Reference Semantics

- Mutable Data Types: list, set, dictionary
  - `x = [0]`



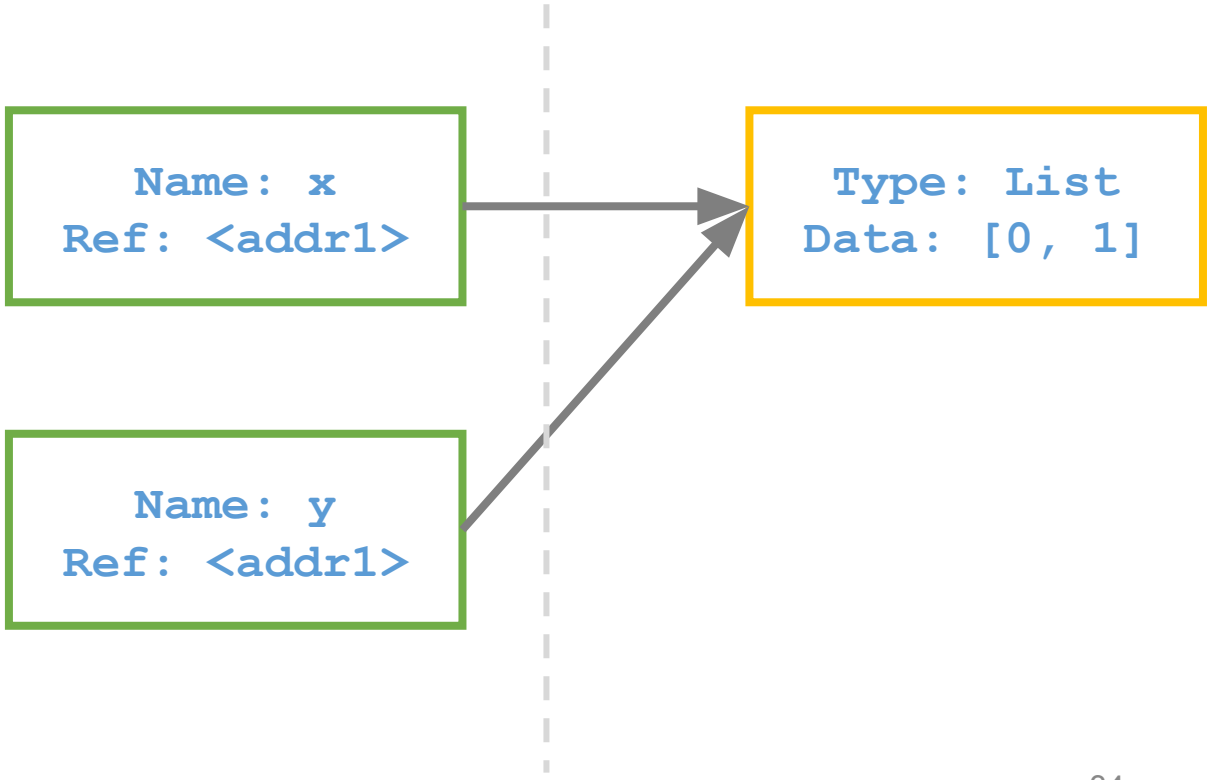
Understanding Reference Semantics

- Mutable Data Types: list, set, dictionary
  - `x = [0]`
  - `y = x`



Understanding Reference Semantics

- Mutable Data Types: list, set, dictionary
  - `x = [0]`
  - `y = x`
  - `y.append(1)`



Sequence Types:

Tuples, Lists, and Strings

Sequence Types

- Tuple
  - **A simple immutable ordered sequence of items**
  - **Items can be of mixed types, including collection types**
- Strings
  - Immutable
  - Conceptually very much like a tuple
- List
  - Mutable ordered sequence of items of mixed types

Similar Syntax

- All 3 sequence types share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are immutable
  - Lists are mutable
- The operations shown in this section can be applied to all sequence types

Sequence Types

- Tuples are defined using parentheses (and commas).

```
tu = (23, 'abc', 4.56, (2, 3), 'def')
```

- Lists are defined using square brackets (and commas).

```
li = ["abc", 34, 4.34, tu, 23]
```

Sequence Types

- Strings are defined using quotes (“, ‘, or “””).

```
st = "Hello World"

s2 = 'Hello World'

s3 = """Hello
World
!
"""
```

Sequence vs Set

	Duplicate	Order
Sequence	allowed	ordered
Set	not allowed	not ordered

Indexing

- We can access individual members of sequence types, using square bracket “array” notation.

```
li = ["abc", 34, 4.34, (2, 3)]
print(li[1])
print(li[-1])
print(li[2:])
```

```
34
(2, 3)
[4.34, (2, 3)]
```

Indexing

- Positive index: count from the left (starting with 0).
- Negative index: count from the right (starting with -1).

```
li = ["abc", 34, 4.34, (2, 3)]
print(li[1])
print(li[-1])
print(li[2:])
```

```
34
(2, 3)
[4.34, (2, 3)]
```

Indexing

- Slicing: return a copy of a subset. `Li[2:]` returns a subset, from second element to the end (more details in Numpy lecture).

```
li = ["abc", 34, 4.34, (2, 3)]
print(li[1])
print(li[-1])
print(li[2:])
```

```
34
(2, 3)
[4.34, (2, 3)]
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

The ‘in’ Operator

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- The `in` keyword is also used in the syntax of for loops and list comprehensions.

The + Operator

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- The * operator produces a new tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> "Hello" * 3
'HelloHelloHello'
```

Mutability

Tuples vs. Lists

Tuples: Immutable

- You can't change a tuple. Instead, you can make a fresh tuple and assign its reference to a previously used name.

```
t = (23, 'abc', 4.56, (2, 3, 'def'))
t[2] = 3.14
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-d7e6ace4c8dc> in <module>
      1 t = (23, 'abc', 4.56, (2, 3, 'def'))
----> 2 t[2] = 3.14

TypeError: 'tuple' object does not support item assignment
```

```
t = (23, 'abc', 3.14, (2, 3, 'def'))
```

Lists: Mutable

- We can change lists in place.

```
li = ['abc', 23, 4.34, 23]
li[1] = 45
print(li)
```

```
['abc', 45, 4.34, 23]
```

- Name `li` still points to the same memory reference when we're done.
- The mutability of lists means that they aren't as fast as tuples.

Tuples vs. Lists

- Lists are slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.
- To convert between two, use the `list()` and `tuple()` functions:
  - `li = list(tu)`
  - `tu = tuple(li)`

Operations on Lists Only

- `li.append('a')`
- `li.insert(2, 'i')`
- `li.extend([9, 8, 7])`
- `li.remove('b')`
- `li.reverse()`
- `li.sort()`
- `li.sort(some_function)`

Dictionary

Dictionary: A Mapping Type

- Dictionaries store a mapping between a set of keys and a set of values.
  - Keys can be any immutable type
  - Values can be any type
  - A single dictionary can store values of different types
- You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.

Using dictionaries

- Pairs value can be accessed by key name:

```
dictionary[pair_key]
```

```
d = {'user' : 'bozo', 'pswd' : 1234}
print(d['user'])
print(d['pswd'])
print(d['bozo'])
```

```
bozo
1234
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-bbb610c0cd80> in <module>
      2 print(d['user'])
      3 print(d['pswd'])
----> 4 print(d['bozo'])

KeyError: 'bozo'
```

Using dictionaries

- We can get list of keys, values and pair(as tuple) with

`keys(), values(), items()`

```
d = {'user': 'bozo', 'pswd': 1234}
print(d.keys())
print(d.values())
print(d.items())
```

```
dict_keys(['user', 'pswd'])
dict_values(['bozo', 1234])
dict_items([('user', 'bozo'), ('pswd', 1234)])
```

Using dictionaries

- We can delete a pair with `del` statement, and clear all pairs with `clear()`.

```
d = {'user': 'bozo', 'pswd': 1234, 'id' : 45}

del d['pswd']
print(d)

d.clear()
print(d)
```

```
{'user': 'bozo', 'id': 45}
{}
```

Functions

Functions

- `def` creates a function and assigns it a name
- `return` **sends a result back to the caller**
- **Arguments and return types are not declared**

```
def <name>(arg1, arg2, ..., argN):
  <statements>
  return <value>

def times(x,y):
  return x*y
```

Passing Arguments to Functions

- Arguments are passed by assignment
- Passed arguments are assigned to local names
- Assignment to argument names don't affect the caller
- Changing a mutable argument may affect the caller

```
def changer (x,y):
  x = 2                    # changes local value of x only
  y[0] = 'hi'              # changes shared object
```

Optional Arguments

- **Can define defaults for arguments that need not be passed**

```
def func(a, b, c=10, d=100):
    print(a,b,c,d)

>> func(1,2)
1 2 10 100
>> func(1,2,3,4)
1 2 3 4
```

Notes

- **All functions in Python have a return value – even if no return line inside the code.**
- **Functions without a return return the specual value `None`.**
- **There's no function overloading in Python**
  - **Two different functions can't have the same name, even if they have different arguments.**

Notes

- **Functions can be used as any other data type. They can be:**
  - **Arguments to function**
  - **Return values of functions**
  - **Assigned to variables**
  - **Parts of tuples, lists, etc**

```
x = lambda a, b : a * b
print(x(5, 6))


def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Control of Flow

if, elif, else

```
def check_sign(x):
    answer = "given value is "
    if x > 0:
        answer += "positive"
    elif x < 0:
        answer += "negative"
    else:
        answer += str(0)
    print(answer)

check_sign(2)
check_sign(-4)
check_sign(0)
```

```
given value is positive
given value is negative
given value is 0
```

while, for

```
n = 2015
div = 2
while n % div != 0:
        div = div + 1
print "Smallest divisor of", n, "is", div
```

```
partial_sum = 0
lst = range(1,101)
for num in lst:
            partial_sum = partial_sum + num
print "The sum is", partial_sum
```

break, continue

```
for elem in lst:
    if elem < 0:
        print "First negative number is", elem
        break
```

```
lst = [1,4,5,8,3,5,7,1,2]
uniques = []
for x in lst:
    if x in uniques:
        continue
    uniques.append(x)
print uniques
```

try, except, finally

```
>>> try:
...     1 / 0
... except:
...     print('That was silly!')
... finally:
...     print('This gets executed no matter what')
...
That was silly!
This gets executed no matter what
```

Recursive Function

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

Modules

Why Use Modules?

- Code reuse
  - Routines can be called multiple times within a program.
  - Routines can be used from multiple programs
- Namespace partitioning
  - Group data together with functions used for that data
- Implementing shared services or data
  - Can provide global data structure that is accessed by multiple subprograms

Modules

- **Modules are functions and variables defined in separate files**
- **Item are imported using `from` or `import`**
  - `from module import function` … call `function()`
  - `import module` … call `module.function()`
- **Modules are namespaces**
  - **Can be used to organize variable names, i.e.** `atom.position = atom.position - molecule.position`

Class and Objects

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

Example: Atom Class

```
class atom(object):
    def __init__(self, atno, x, y, z):
        self.atno = atno
        self.position = (x, y, z)
    def get_position(self):
        return self.position
    def __repr__(self):
        return '%d %10.4f %10.4f %10.4f' %
(self.atno, self.position[0], self.position[1],
self.position[2])

at = atom(6, 0, 1, 2)
print(at)
```

```
6      0.0000      1.0000      2.0000
```

Example: Atom Class

- **Overloaded the default constructor** (and print operator)
- **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**
- We now want to use the atom class to build molecules …

Example: 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 moecule named %s\n' % self.name
        str = str + 'It has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            str = str + repr(atom) + '\n'
        return str
```

Example: 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 moecule named Water
It has 3 atoms
8     0.0000     0.0000     0.0000
1     0.0000     0.0000     1.0000
1     0.0000     1.0000     0.0000
```

Inheritance

```
class monoatomic_molecule(molecule):
    def num_of_molecule(self):
        return len(self.atomlist)

    def addatom(self, atom):
        if len(self.atomlist) == 0:
            super(monoatomic_molecule, self).addatom(atom)
        else:
            for at in self.atomlist:
                if at.atno != atom.atno:
                    print("This molecule is monoatomic!")
                    return
            super(monoatomic_molecule, self).addatom(atom)
```

Inheritance

```
monomol = monoatomic_molecule('Ozone')
monomol.addatom(atom(8, 1, 0, 0))
monomol.addatom(atom(8, 0, 1, 0))
monomol.addatom(atom(8, 0, 0, 1))

monomol.addatom(atom(2, 0, 0, 1))

print(monomol)
```

Inheritance

```
This molecule is monoatomic!
This is a moecule named Ozone
It has 3 atoms
8     1.0000     0.0000     0.0000
8     0.0000     1.0000     0.0000
8     0.0000     0.0000     1.0000
```

Inheritance: Monoatomic Molecule

- `__init__` and `__repr__` are taken from the parent class (molecule)
- Added a new function num_of_molecule()
- Example of code reuse
  - Basic functions don't have to be retyped, just inherited
  - Less to rewrite when specifications change

Overloading

- We defined a new version of `addatom(self, atom)`. Now the function checks whether passed atom object has same atno with existing atoms in `self.atomlist`.
- Also, we extended the parent function by reusing it with `super(monoatomic_molecule, self).addatom(atom)` method.

Public and Private Data

- 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