

Introduction to Python

Instructor

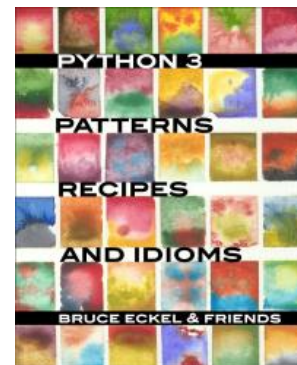
Battista Biggio

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence

University of Cagliari, Italy

Python

- Open source general-purpose language
- Object Oriented, Procedural, Functional
- Easy to interface with C/C++/ObjC/Java/Fortran
- Great interactive environment
- Downloads: <https://www.python.org>
- Documentation: <https://docs.python.org/>
- In this course: **Python 3**
 - For a list of the main changes from Python 2.7 to Python 3:
 - https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- Many available Python books/courses online
 - <https://python-3-patterns-idioms-test.readthedocs.io/en/latest/index.html>
 - <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>

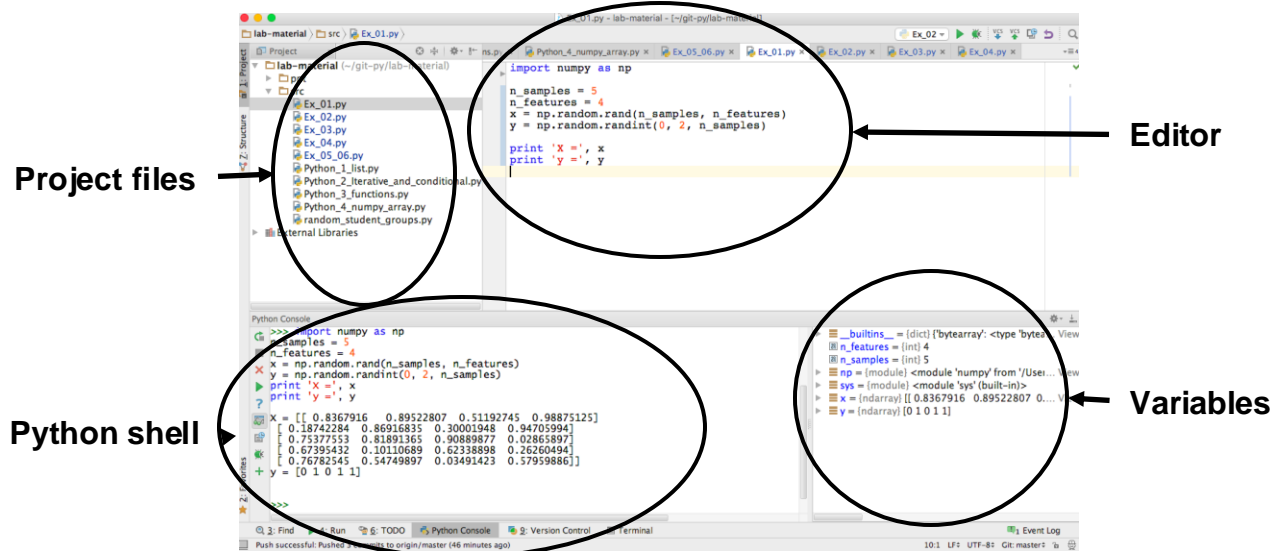


Why Python?



- Reasonable trade-off between a high-level language abstraction and computational efficiency
 - C/C++ is more efficient, but requires implementing at a lower abstraction level (e.g., considering memory allocation)
 - Matlab is maybe easier to program, but typically less efficient
- Python is a high-level programming language providing several third-party libraries that normally wrap C/C++ (efficient) code
- Availability of many third-party libraries (sklearn, tensorflow, pytorch, opencv)

- Development IDE <https://www.jetbrains.com/pycharm/>
- Easy to setup and use
 - Integration with repositories / versioning mechanisms (subversion, git)
- Compliant to PEP8 programming guidelines (after configuration)



The Python Interpreter

- Interactive interface to Python

```
bat:~ bat$ python
Python 2.7.13 |Continuum Analytics, Inc.| (default, Dec 20 2016, 23:05:08)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> 2 * 3
6
>>> exit()
bat:~ bat$
```

- Python interpreter evaluates inputs:
 >>> 2*3
 6
- Python prompts with '>>>'
- To exit Python: press CTRL-D or type exit()

Running Programs on UNIX

% python filename.py

- You could make the *.py file executable (e.g., `chmod +x filename.py`) and add the following `#!/usr/bin/env python` to the top to make it runnable

Built-in Libraries Available

- Large collection of proven modules included in the standard distribution
- <https://docs.python.org/3.7/py-modindex.html>

Numpy (Scipy)

- Offers Matlab-like capabilities within Python Fast array operations
- 2D arrays, multi-dimensional arrays, linear algebra etc.
- Website: <http://www.numpy.org>
- Tutorial: <https://docs.scipy.org/doc/numpy/user/quickstart.html>

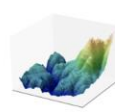
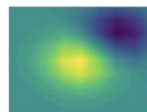
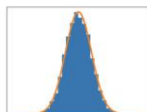
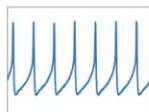
Matplotlib

- Plotting library
- Website: <https://matplotlib.org>



[home](#) | [examples](#) | [tutorials](#) | [API](#) | [docs](#) »

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and **IPython** shells, the **Jupyter** notebook, web application servers, and four graphical user interface toolkits.



Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. For examples, see the [sample plots](#) and [thumbnail gallery](#).

For simple plotting the `pyp1ot` module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Python Basics

A Code Sample

```
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
- The basic printing command is `print`
- 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 (default for numbers)**

```
z = 5 // 2    # Answer is 2, integer division (new in Python 3!)  
z = 5 / 2     # Answer is 2.5 (this returns 2 in Python 2 - cast to int)
```

- **Floats**

```
z = 3.456
```

- **Strings**

- Can use “” or ’ to specify.
“abc” ‘abc’ (Same thing.)
- Unmatched can occur within the string.
“matt’s”
- Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
“””a ‘b”c”””

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 to next line prematurely
- No braces `{ }` to mark blocks of code in Python... Use *consistent* indentation instead
 - 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)

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.
- The development environment, debugger, and other tools use it
 - it’s good style to include one

```
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
- **You create a name the first time it appears on the left side of an assignment expression:**
`x = 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 properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
```

```
Traceback (most recent call last):  
  File "<pyshell#16>", line 1, in -  
    toplevel- y  
NameError: name 'y' is not defined  
>>> y = 3  
>>> y  
3
```

Multiple Assignment

- You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
```

```
>>> x
```

```
2
```

```
>>> y
```

```
3
```

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

Understanding Reference Semantics in Python

Understanding Reference Semantics

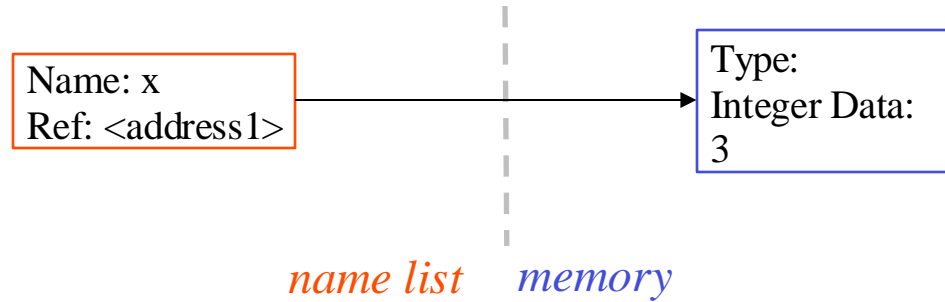
- **Assignment manipulates references**
 - `x = y` **does not make a copy** of the object `y` references
 - `x = y` makes `x` **reference** the object `y` references
- **Very useful; but beware!**
- **Example:**

```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a           # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b          # if we print what b references,
[1, 2, 3, 4]         # SURPRISE! It has changed...
```

Why?

Understanding Reference Semantics

- There is a lot going on when we type:
 - $x = 3$
- First, an integer **3** is created and stored in memory
- A name ***x*** is created
- A *reference* to the memory location storing the **3** is then assigned to the name ***x***
- So, when we say that the value of ***x*** is **3**, we mean that ***x*** now refers to the integer **3**



Understanding Reference Semantics

- The data 3 we created is of type integer. In Python, the datatypes **integer**, **float**, and **string** (and **tuple**) are “**immutable**”
- This doesn't mean we can't change the value of x
 - i.e., change what x refers to ...
- For example, we could increment x:

```
>>> x = 3
```

```
>>> x = x + 1
```

```
>>> print x
```

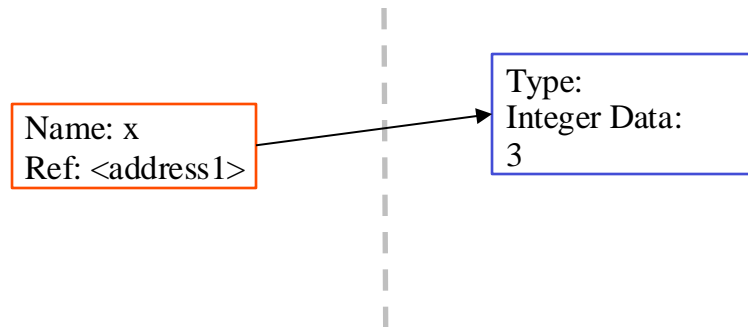
```
4
```

Understanding Reference Semantics

- If we increment x , then what's really happening is:

1. *The reference of name x is looked up.*
2. *The value at that reference is retrieved.*

```
>>> x = x + 1
```

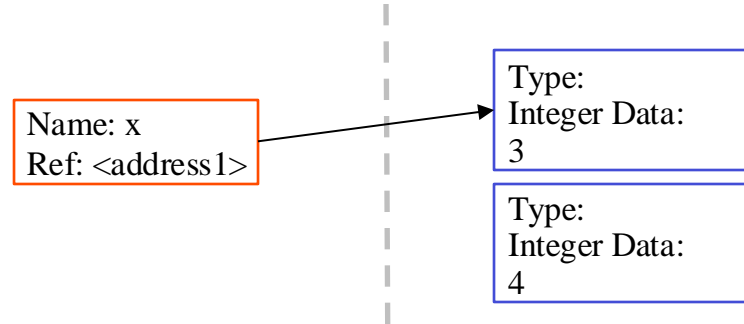


Understanding Reference Semantics

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.
3. *The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.*

>>> $x = x + 1$

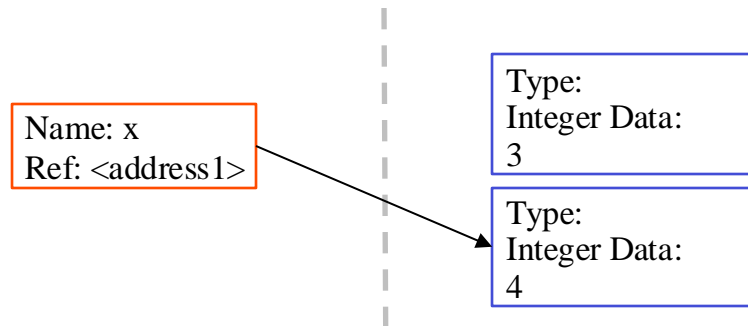


Understanding Reference Semantics

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.
3. The $3+1$ calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
4. The name x is changed to point to this new reference.

>>> $x = x + 1$

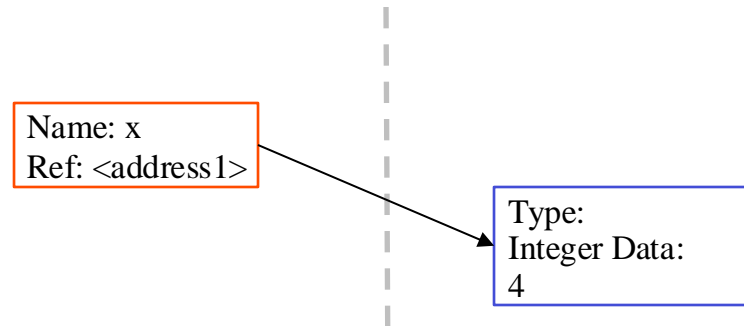


Understanding Reference Semantics

- If we increment x , then what's really happening is:

1. The reference of name x is looked up.
2. The value at that reference is retrieved.
3. The $3+1$ calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
4. The name x is changed to point to this new reference.
5. The old data 3 is garbage collected if no name still refers to it.

>>> $x = x + 1$



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

```
>>> x = 3      # Creates 3, name x refers to 3
>>> y = x      # Creates name y, refers to 3.
>>> y = 4      # Creates ref for 4. Changes y.
>>> print x    # No effect on x, still ref 3.
3
```

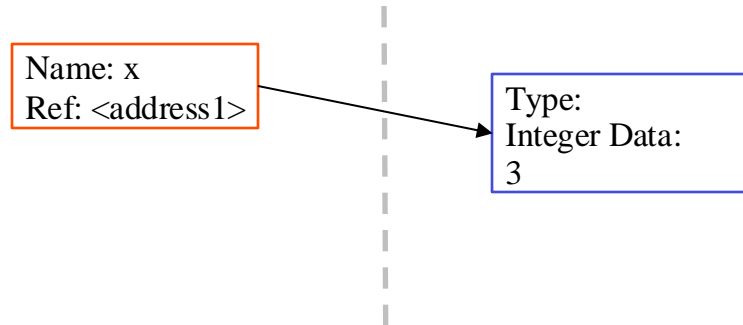
|
|
|
|
|
|
|
|
|

Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

→

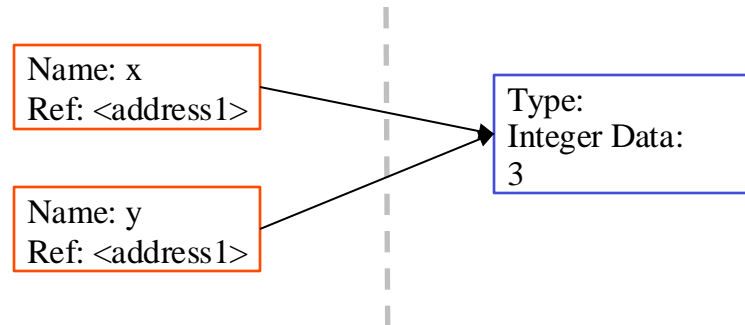
```
>>> x = 3      # Creates 3, name x refers to 3
>>> y = x      # Creates name y, refers to 3.
>>> y = 4      # Creates ref for 4. Changes y.
>>> print x    # No effect on x, still ref 3.
3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

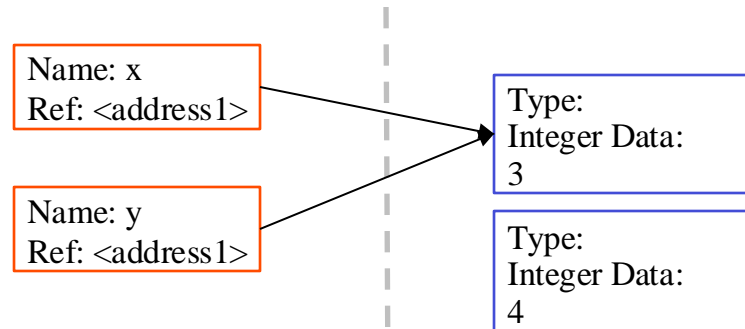
```
→ >>> x = 3      # Creates 3, name x refers to 3
    >>> y = x      # Creates name y, refers to 3.
    >>> y = 4      # Creates ref for 4. Changes y.
    >>> print x    # No effect on x, still ref 3.
    3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

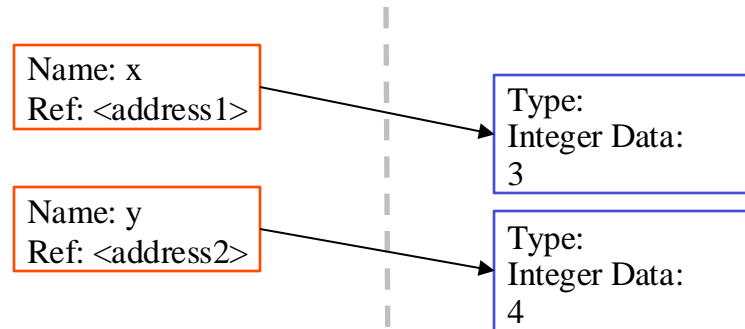
```
>>> x = 3      # Creates 3, name x refers to 3
>>> y = x      # Creates name y, refers to 3.
→ >>> y = 4    # Creates ref for 4. Changes y.
>>> print x    # No effect on x, still ref 3.
3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

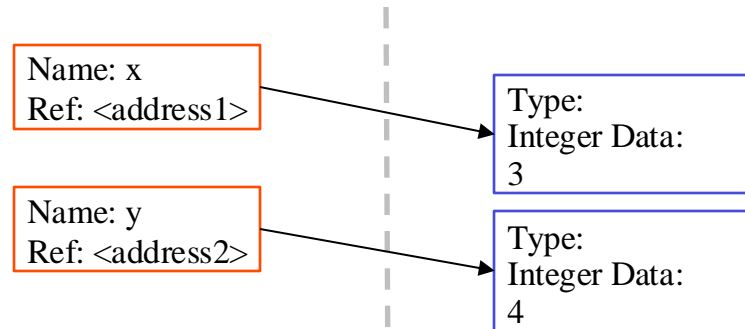
```
>>> x = 3      # Creates 3, name x refers to 3
>>> y = x      # Creates name y, refers to 3.
→ >>> y = 4    # Creates ref for 4. Changes y.
>>> print x    # No effect on x, still ref 3.
3
```



Assignment 1

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as one expects:

```
>>> x = 3      # Creates 3, name x refers to 3
>>> y = x      # Creates name y, refers to 3.
>>> y = 4      # Creates ref for 4. Changes y.
→ >>> print x  # No effect on x, still ref 3.
3
```



Assignment 2

- For other data types (lists, dictionaries, user-defined types), assignment works differently.
 - These data types are “**mutable**”
 - When we change these data, we do it *inplace*
- We do not copy them into a new memory address each time
- If we type `y=x` and then modify `y`, both `x` and `y` are changed

Immutable

```
>>> x=3
>>> y=x
>>> y=4
>>> print x
3
```

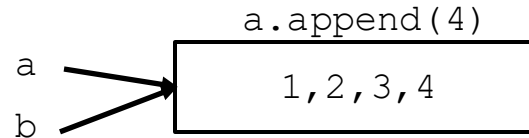
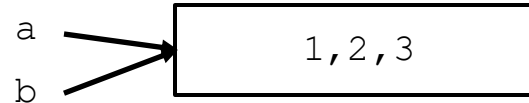
Mutable

```
x = some mutable object
y = x
make a change to y look at x
x will be changed as well
```

Why? Changing a Shared List

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

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



Our Surprising Example No Longer Surprising...

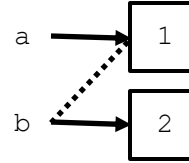
- So now, here's our code:

```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a           # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b         # if we print what b references,
[1, 2, 3, 4]        # SURPRISE! It has changed...
```

Recap: Mutable vs Immutable Objects

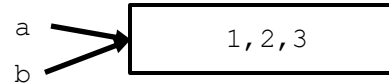
- Immutable objects can not be modified (reference changes!)

```
>>> a = 1
>>> b = a
>>> b = 2
>>> print a, b
1 2
```

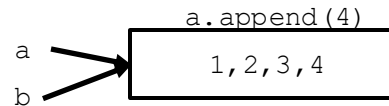


- Mutable objects (like *lists*) can be modified (by *in-place* functions)

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



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



Sequence types: Tuples, Lists, and Strings

Sequence Types

1. Tuple

- A simple *immutable* ordered sequence of items
- Items can be of mixed types, including collection types

2. Strings

- *Immutable*
- Conceptually very much like a tuple

3. List

- *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- **Key differences:**
 - Tuples and strings are immutable
 - Lists are mutable
- The operations shown in this section can be applied to *all* sequence types
 - most examples will just show the operation performed on one

Sequence Types 1

- **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, 23]
```

- **Strings are defined using quotes (" , ' , or """).**

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line  
string that uses triple  
quotes."""
```

Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- **Note that all are 0 based...**

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1] # Second item in the tuple.
'abc'
```

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1] # Second item in the list.
34
```

```
>>> st = "Hello World"
>>> st[1] # Second character in string.
'e'
```

Positive and Negative Indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]  
'abc'
```

Negative lookup: count from right, starting with -1.

```
>>> t[-3]  
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying 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))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]  
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]  
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

To make a **copy** of an entire sequence, you can use `[:]`.

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1      # 2 names refer to 1 ref
                        # Changing one affects both
```

```
>>> list2 = list1[:]   # Two independent copies, two refs
```

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
```

- **For strings, tests for substrings**

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

- **Be careful: 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

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

```
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-  tu[2] =
    3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

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

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place*.
- 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.

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, 'a']

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

The *extend* method vs the **+** operator

- **+** creates a fresh list (with a new memory reference)
- *extend* operates on list *li* in place

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- *Confusing:*
 - Extend takes a list as an argument.
 - Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> 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 4

```
>>> 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
```


Recap of Basic Operations (Lists)

- Creation

```
>>> a = [1, 2, 3]
>>> a = range(0,5)
>>> print a
[0, 1, 2, 3, 4]
>>> a = range(0,5,2)
>>> print a
[0, 2, 4]
```

- Indexing

```
>>> a = [1, 2, 3]
>>> print a[0], a[1], a[-1]
1 2 3
>>> print a[1:] # slicing from index 1 to end
[2,3]
>>> print a[:2] # slicing from index 0 to 2 (excluded)
[1,2]
```

Recap of Basic Operations (Lists)

- Reference and copy

```
>>> a = [1, 2, 3]
>>> b = a # reference assignment. a and b are modified
>>> a[0] = 2
>>> print a, b
[2, 2, 3] [2, 2, 3]

>>> b = a[:] # this is a copy. now a and b are different
>>> a[0] = 2
>>> print a, b
[2, 2, 3] [1, 2, 3]
```

- Other operations

```
>>> a = [3, 1, 2]
>>> len(a) # returns number of elements in list
3
>>> a.sort() # sorts list in ascending order [1, 2, 3]
>>> a.append(4) # appends element at end -> [1, 2, 3, 4]
```

Tuples vs. Lists

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

Dictionaries

Dictionaries: A Mapping Type

- **Dictionaries store a mapping between a set of keys and a set of values**
 - Defined within *braces* {...}
 - `a = { 'key1': 0, 'key2': 1 }`
- 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* (or *items*) in the dictionary.**

Using Dictionaries

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

>>> d = {'user': 'bat', 'pswd': 1234}
>>> d['user'] = 'clown'
>>> d
{'user': 'clown', 'pswd': 1234}

>>> d['id'] = 45
>>> d
{'user': 'clown', 'id': 45, 'pswd': 1234}
```

Using Dictionaries

```
>>> d = {'user': 'bat', 'p': 1234, 'i': 34}
>>> del d['user'] # remove 'user'
>>> d
{'p': 1234, 'i': 34}

>>> d.clear() # remove all
>>> d
{}
```

Using Dictionaries

```
>>> d = {'user': 'bat', 'p': 1234, 'i': 34}
```

```
>>> d.keys()    # List of keys  
['user', 'p', 'i']
```

```
>>> d.values()  # List of values  
['bat', 1234, 34]
```

```
>>> d.items()   # List of item tuples  
[('user', 'bat'), ('p', 1234), ('i', 34)]
```


Functions

Functions

- *def* creates a function and assigns it a name
- *return* sends a result back to the caller
- Arguments are passed by assignment
- 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
- Assignments 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 *optional* arguments

```
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
```

More on Functions

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

Control of Flow

Control of Flow

- **if** *condition*:
 statements
elif *condition*:
 statements
else:
 statements
- **for** *var in sequence*:
 statements
- **while** *condition is True*:
 statements

```
a=0
if a<0:
    print 'a is negative'
elif a>0:
    print 'a is positive'
else:
    print 'a is zero'
```

```
a=['a','b','c']
for elem in a:
    print elem
```

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

Numpy Arrays

Numpy Arrays (array)

```
import numpy as np

a = np.array([[1, 2, 3],
              [4, 5, 6]])

>>> a.shape
(2, 3)

>>> a.dtype
dtype('int64')

n_rows, n_cols = 2, 4
a = np.zeros(shape=(n_rows, n_cols))
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Other Array Creation Methods

```
a = np.ones(shape=(n_rows, n_cols)) # creates matrix of ones
```

```
a = np.eye(n_rows, n_cols) # creates identity matrix
```

```
>>> a
```

```
array([[ 1.,  0.,  0.,  0.],  
       [ 0.,  1.,  0.,  0.]])
```

```
# random numbers from Normal distribution
```

```
# with zero mean and unit variance
```

```
a = np.random.randn(n_rows, n_cols)
```

```
# random numbers from Uniform distribution in [0,1]
```

```
a = np.random.rand(n_rows, n_cols)
```

```
>>> a
```

```
[[ 0.02413213  0.27179266  0.8904404  0.78110291]  
 [ 0.66329305  0.57972581  0.19168138  0.44013989]]
```

```
>>> np.random.randint(0, 5, [n_rows, n_cols]) # random integers
```

```
[[ 0  2  1  0]  
 [ 1  4  3  1]]
```

Array Indexing

- This can be rather complicated
 - <https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html>
- Let's keep it simple, using some simple rules
 - Index arrays with shape = (n,) (flat arrays, vectors) with a single index
 - Index matrices with shape = (n, m) using two indices
 - In general, if shape.size == K, we should use K indices

```
a = np.eye(3)
>>> a
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> a[0, 0] # picks the first element (returns float)
1.0
>>> a[0, :] # picks the first row (returns flat array)
array([ 1.,  0.,  0.])
>>> a[:, 1] # picks the second column (returns flat array)
array([ 0.,  1.,  0.])
```

Array Indexing

```
a = np.eye(3)
>>> a
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> a[0:2, 0:2] #selects submatrix with slicing operators
array([[ 1.,  0.],
       [ 0.,  1.]])

b = np.array([1, 1, 0])
>>> a[b==1, :] # b used to index rows (picks first and second here)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

Other Operation on Arrays

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

>>> np.vstack((a, b)) # stack rows (vertical stacking)
array([[1, 2, 3],
       [4, 5, 6]])

>>> np.hstack((a, b)) # stack columns (horizontal stacking)
array([1, 2, 3, 4, 5, 6])

>>> a.dot(b) # scalar product
32

# other operations include: reshape, transpose, min/max, etc.
# we will see more throughtout the course, when required
```