Python Pluses and Minuses

● The Good:
  ○ Fast Prototyping
    ■ More of a high level language so you can spend more time completing your task
  ○ Lots of scientific computing packages (NumPy, SciPy, scikit-learn, nltk, PyRosetta, BioPython)
  ○ Versatile
    ■ Can be used for writing small shell scripts, data analysis, running a website, etc.

● The Bad:
  ○ Can be slow
  ○ Type system makes it hard to manage large projects & creates unexpected bugs (more on this later!)
A Note on Versions

- In 2020, at long last, **Python2** will no longer be supported and the world will have to transition to **Python3**
- This class will use and support **Python2**!
- If you only know **Python3**, they are very similar, so don’t worry!
  - Key Differences:
    - Keyword `print` vs. function `print()`
    - `range()` and `zip()` return an object of type list
    - **INTEGER DIVISION ROUNDS DOWN** (e.g. `3 / 2 = 1` in Python2)
Quickstart Guide to Python Syntax
Variables

\[ x = 279 \]

\[ \text{type}(x) \text{ returns } \text{<type 'int'>} \]

\[ f = 279.0 \]

\[ \text{type}(f) \text{ returns } \text{<type 'float'>} \]

\[ s = 'this is a string' \]

\[ \text{type}(s) \text{ returns } \text{<type 'str'>} \]

\[ b = \text{True} \]

\[ \text{type}(b) \text{ returns } \text{<type 'bool'>} \]

Single or double quotes are both fine, there is no separate idea of characters.

In some other languages, True and False are lower case!

Use str(), int(), float() etc. to convert between types.
Variables Require Constant Vigilance!

- Notice how we didn’t have to explicitly declare a type for our variables
  - Easy to forget what type something is
  - Even worse: you can reuse a variable name for a different type
    - e.g. `x = 5` followed by `x = 'five'`
  - When in doubt, call `type()` on a variable to see its type.
  - Fun Fact: calling `type(type('s'))` returns `<type 'type'>`!
Strings

- **Concatenation**
  - `my_str = 'good'`
  - `len(my_str) == 4`
  - `my_str += ' job'`
  - `my_str == 'good job'`
  - `len(my_str) == 8`

- **Supports indexing**
  - `my_str[0] == 'g'`
  - `my_str[1] == 'o'`
  - `my_str[0:4] == 'good'`
  - `my_str[-1] == 'b'`

- **Immutable**
  - `my_str[7] = 'g'`
  - This will throw an `error` since strings cannot be modified in place
residue = 'ALA'
if residue == 'ASP':
    print 'The pKa of {} is {}'.format(residue, 3.65)
elif residue == 'LYS':
    print 'The pKa of {} is {}'.format(residue, 10.53)
else:
    print residue
    print 'I don\'t care about the other residues.'

Handy way to handle string formatting & avoid type errors!
Other languages use “else if” instead of “elif”

Note: Python doesn’t use braces to keep track of code blocks, it uses whitespace!

Every line in the code block needs to have the same indentation. Beware of mixing tabs and spaces!
residue = 'ALA'
if residue == 'ALA':
    print residue
else:
    print 5 + '5'

Python is an interpreted language: it doesn't compile the source code ahead of time into a separate file.

It "lightly" checks for errors and then only looks at what it actually executes.

print 5 + '5' is erroneous but will never throw an error since the code is never executed.
Lists

- Ordered, mutable container, can hold objects of different types
- Initialization:
  
  ```
  residues = list()
  residues = []
  residues = ['ALA']
  ```
- Useful commands:
  ```
  residues.append('ARG')
  residues[0] == 'ALA'
  residues[1] == 'ARG'
  len(residues) == 2
  residues.pop()
  residues == ['ALA']
  residues[0] = 'TYR'
  ```
- Supports slicing:
  ```
  residues = ['ALA', 'ARG', 'ASN', 'ASP']
  residues[1:3] == ['ARG', 'ASN']
  residues[2:] == ['ASN', 'ASP']
  ```
More Lists

- Can have lists of lists
  - `a = [1, 2]`
  - `b = [3, 4]`
  - `c = [a, b]`
  - `c[0][0] == 1`
  - `c[1][1] == 4`

- List comprehensions
  - We'll get to these once we cover looping!
Dictionaries

● Unordered mapping from keys to values

● Initialization
  ○ resmap = dict()
  ○ resmap = {}
  ○ resmap = {'aliphatic': 'ALA', 'alcohol': 'SER'}

● Useful commands
  resmap.keys() == ['aliphatic', 'alcohol']
  resmap.values() == ['ALA', 'SER']
  len(resmap) == 2

● Accessing a key that doesn't exist (e.g. resmap['charged']) throws an error!
  ○ If you are feeling fancy & have some spare time, look into the defaultdict!

Caveat: assumed the ordering for keys() and values() for simplification
Tuples

- Like Lists, except they are immutable
  - Used to hold things you know will never change
  - Used as keys for dictionaries since keys must be hashable (lists are not hashable in Python)

- Initialization
  - `tup = tuple()`
  - `tup = ()`
  - `tup = (1, 2, 3)`
  - **BE CAREFUL**: `tup = (3)` will create an integer, since Python confuses `()` with math notation here. Use `tup = (3,)` to create a singleton tuple.
Looping

For Lists/Tuples:
residues = ['ALA', 'ARG', 'ASN']
for res in residues:
    print res

for i in range(len(residues)):
    print residues[i]

For Dictionaries:
resmap = {'aliphatic': 'ALA', 'alcohol': 'SER'}
for k in resmap:
    print k, resmap[k]
List Comprehensions

Old Way:

```python
mylist = []
for i in range(1000000):
    mylist.append(i % 2)
```

New Way:

```python
mylist = [i % 2 for i in range(1000000)]
```

Sometimes faster & always cooler!

~.3 seconds vs. ~.2 seconds on my laptop for this toy example!
Other List Functions

- **sum()**
  - sum(range(1, 101)) == sum([i for i in range(1, 101)]) == 5050

- **sorted()**
  - sorted([5, 10, 8]) == [5, 8, 10]
  - sorted([5, 10, 8], reverse=True) == [10, 8, 5]
  - sorted([5, 10, 8], key=lambda x: abs(7 - x)) == [8, 5, 10]

- **map()**
  - map(lambda x: x**2, [1, 2, 3]) == [1, 4, 9]

- **filter()**
  - filter(lambda x: x % 5 == 0, [5, 10, 279]) == [5, 10]

Naming a variable sum is a really bad idea because it overwrites this function, making it unusable in the future!

Custom sorting function!
Sort by numbers closest to 7
Importing

- When you need to access code outside the current file!

```python
import time
tic = time.time()
sum([i for i in range(1, 100000)])
toc = time.time()
print toc - tic

from math import sin, pi
print sin(pi)
```
Functions

def compare_strs(s1, s2):
    return s1 < s2

compare_strs('aardvark', 'zygomatic')

import os
def where_am_I():
    print os.getcwd()

where_am_I()
Classes

```python
class Predictor(object):
    def __init__(self, nIters, name):
        self.nIters = nIters
        self.name = name

    def predict(self, start):
        raise NotImplementedError("Predictor should not be instantiated")

class MonteCarloPredictor(Predictor):
    def predict(self, start):
        # Do some stuff
        for x in self.nIters:
            # do some stuff
            pass

myPredictor = MonteCarloPredictor(nIters=1000, name='myPred')
myPredictor.predict(pose)
```
NumPy

- Numerical computing library for Python designed to be efficient.
- Great for linear algebra and data processing tasks

```python
import numpy as np
x = np.arange(2)
y = np.arange(2, 4)
np.dot(x, y) computes the dot product of x and y

A = np.arange(4).reshape((2, 2))
np.matmul(A, x) does matrix multiplication between A and x
np.linalg.eig(A) returns the eigenvalues and eigenvectors of A
```