Introduction to Python Programming

UCSD ECE SUMMER WORKSHOP

```
COURSE_0 = ["8-25-2015", "8-27-2015"]
COURSE_1 = ["9-02-2015", "9-03-2015"]
```

Overview

- Introduction
- Python as a calculator
- Basics
  - Variables
  - Types
  - Control flow
- Functions
- Data structures
- Importing libraries
- Practical example: image processing
- Practical example: anagram detection

Introduction

What is Python?

- Python is a high-level programming language which will allow you to get a lot of work done with few lines of code.
- It is supported by many modern operating systems (Windows, Mac, Linux, Android, iOS, etc.)
- Since it is an interpreted language, there is no need to compile your code, as you would in C/C++/Java/etc.
- Surprisingly, Python is named not after the snake, but rather after the comedy troupe **Monty Python**

Why use Python?

- Because it is supported by so many platforms, your code will not be limited to just your machine, or even just your operating system!
- There is already massive support for python in the form of third-party libraries. These will help you do things like:
  - Develop a game using PyGame
  - Make really cool plots for your conference paper using matplotlib
  - Build a graphical user interface (GUI) using PyQt
  - Scientific computing using NumPy
  - Web development using Django
  - The list goes on…
- Python's flexibility allows you to be massively productive without writing too many lines of code.
- Python bindings are available for many popular APIs: Google services, Amazon AWS, etc.

Two branches

v2.x (current as of July 2015 is 2.7.10)

- Considered as legacy
- Still the default for many OS

v3.x (current as of July 2015 is 3.4.3)

- Considered as the current version
- Changelog here: https://docs.python.org/3/whatsnew/3.0.html

The code in this class is intended to run on **v2.7.10**

More on the difference between the two branches here:

- https://wiki.python.org/moin/Python2orPython3

Before we start

Hopefully, by this point, everyone has installed Python on their laptop.

I highly recommend that you type alongside me during this course. I think you'll get more out of it that way!

If you want to learn more, there are some really great lectures online.

- Jessica McKellar @ PyCon 2014
- Series from Google Developers (2 days)

If anyone does not have Python installed at this point, please try one of these:

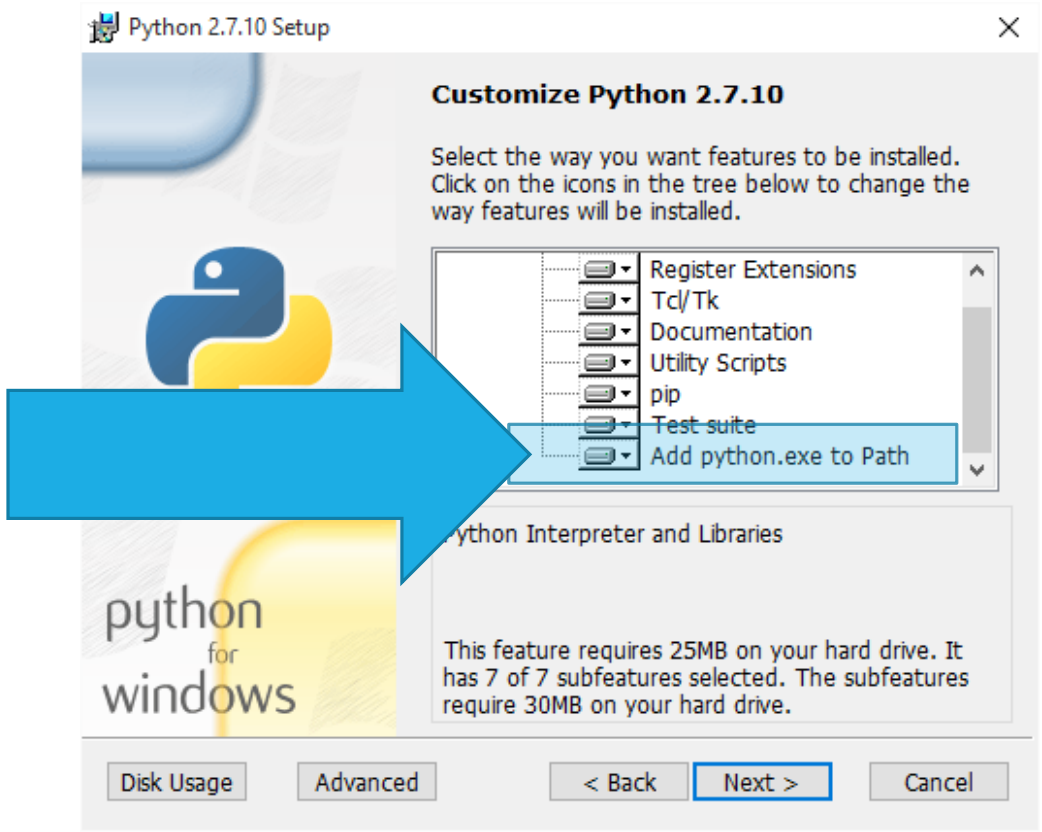
- Go ahead and install from python.org.
- If, for whatever reason, you cannot install Python (perhaps because of access privileges or some other such thing), please use a web-based Python interpreter for this class: Coding Ground.
- As a last resort, use this one: repl.it. It has a nice UI but supports Python 3 instead of 2.

Add to path

When installing Python in Windows, I recommend enabling "**Add python.exe to Path**".

This will enable you to call Python from any directory.

```
# if not added to path
C:\scripts>C:\Python27\python.exe

# if added to path
C:\scripts>python
```



Python Interpreter

Python is a program itself. Usually we pass it files (scripts) that we have written.

Instead, let's open the interpreter directly and have a look around.

Python as a calculator

Python as a calculator

Need to quickly calculate something? Python is a great choice. All of the mathematical operations you expect are built right in.

```
>>> 5 + 3
8

>>> 5.0 + 3.0
8.0

>>> 10 * 3
30

>>> 20 - 5 - 3
12
```

```
>>> x = 5
>>> x -= 3 # same as x = x - 3
>>> print x
2

>>> x += 8 # same as x = x + 8
>>> print x
10
>>> x *= 3  # same as x = x * 3
>>> print x
30
```

Python as a calculator: division

One thing to be careful with is division. Integers and floating point numbers are handled differently for division. If you divide an integer by an integer, Python expects the result to also be an integer, which may not be what you expect.

```
# division between two integers
>>> x = 10 # int
>>> y = 3 # int
>>> x / y
3

# potential pitfall
>>> score = 78
>>> pct = score / 100
>>> print pct
0
```

```
# division between two floats
>>> x = 10.0 # float
>>> y = 3.0 # float
>>> x / y
3.3333333333333335

# division between float and int
>>> pi = 3.14159
>>> pi / 2
1.570795
```

Basics

Variables

Variables let you keep track of things while programming. Without them, everything would be chaotic! Python is very forgiving with variables, which is one of the reasons it is so easy to code in. You don't need to declare a variable or its type. Python will figure it out for you.

```
>>> a = "wall-e"
>>> print a
wall-e

>>> a = 10
>>> print a
10

>>> b = True
>>> print b
True
```

```
>>> score0 = 73.3
>>> score1 = 92.5
>>> mean = (score0 + score1) / 2
>>> print mean
82.9

>>> hiScore = max(score0, score1)
>>> print hiScore
92.5
```

Variables: naming rules

Remember that variables are **case-sensitive**!

¯_(ツ)_/¯

Your variable name must start with a letter or underscore (_).

The remainder of your variable name may contain: letters, numbers and underscores.

A few keywords are reserved for Python. You can't use these as variable names.

Valid	Invalid
x	7x
userName	temp-kelvin
_user001	True
true	001_user
temp_kelvin	for
_	user#tag

Types

There are many others, but this course will be focusing on the integer (**int**), floating point (**float**), string (**str**) and boolean (**bool**) data types. You can check the type of any variable in python using the **type()** function, as below.

```
>>> type(5)
<type 'int'>

>>> type(3.14)
<type 'float'>

>>> type("hello")
<type 'str'>
```

```
>>> type('world')
<type 'str'>

>>> type(True)
<type 'bool'>

>>> type(False)
<type 'bool'>
```

Typecasting

As with other programming languages, variables in python can be **cast** between different **types**. In some cases, this will result in truncation.

- Some types cannot be cast between. For example, float("cat") will return an error.

```
>>> float(5)
5.0

>>> int(3.14)
3

>>> int(3.9999999)
3

>>> str(5)
'5'
```

```
>>> str(3.14)
'3.14'

>>> int("100")
100

>>> float("cat")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to
float: cat
```

Built-in functions

Python has many built-in functions. These are functions that can be called from anywhere in your program, without including any additional libraries. You can find a full list of built-in functions here: https://docs.python.org/2/library/functions.html.

We will be focusing on just a few. Remember these are **case sensitive**!

Function	Description	Category
id()	Determine memory location of object	Informational
input()	Query user for input	Interaction with user
int()	Convert input to int	Typecasting
float()	Convert input to float	Typecasting
len()	Determine the length of a list/string/etc.	Tool
print()	Print something to the user	Interaction with user
range()	Returns a list of integers	Tool
str()	Convert input to string	Typecasting
type()	Determine type of object	Informational

Control flow – range()

The range() function is used to generate a **list** of **integers**. This function is used quite often in for loops, as we will see shortly. range(x) creates the list [0, 1, …, x-2, x-1]. It **does not** contain x!

- Syntax is: range(**[start]**, **stop**, **[step]**) – both **start** and **step** are optional. Range requires only **stop**.
- If **start** is omitted, it will default to 0.
- If **step** is omitted, it will default to +1.

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

>>> range(0, 5)
[0, 1, 2, 3, 4]

>>> range(0, 5, 2)
[0, 2, 4]
```

```
>>> range(-2, 5)
[-2, -1, 0, 1, 2, 3, 4]

>>> range(0, -5)
[]

>>> range(0, -5, -1)
[0, -1, -2, -3, -4]
```

Control flow – if

If allows for code branching. Enter the if loop **if** the condition is true, otherwise fall back to any trailing elif (else if) branches. There may be multiple elif and a single else for any if.

```
# age.py
age = input("enter your age: ")
if age < 15:
    print “y0 d00dz!"
elif age < 30:
    print "Hey!“
elif age < 45:
    print “Good day, sir!”
else:
    print “Salutations!“
# ------------------------------

> age.py
enter your age: 12
y0 d00dz!
```

Containment

A powerful feature of Python is the “in” operator. This will allow you to check if an item is contained in a list or dictionary; or if a character is contained in a string. You can also use “not in” to negate the containment.

```
>>> "y" in "python"
True

>>> "x" in "python"
False

>>> 5 in [1, 3, 5, 7]
True

>>> 2 not in [1, 3, 5, 7]
True

>>> "salt" not in ["salt", "pepper"]
False
```

Control flow – for

The for loop allows you to iterate over a set of items.

- It is commonly used with the “in” containment operator and a list of items.
- The user specifies an iterating variable which is used inside the for loop.
- The range() function is very useful in conjunction with for, as can be seen in the example.

```
# print students in a class
>>> students = ["Al", "Bob", "Carl"]
>>> for student in students:
...     print student
...

Al
Bob
Carl
```

```
# compute sum of numbers 1-10
>>> sum = 0
>>> for x in range(1, 11):
...     sum += x
...

>>> print sum
55
```

Control flow – others

While loop – loop which will continue indefinitely until some condition forces an exit.

Continue – continue with next iteration of loop without completing current iteration.

Break – breaks out of the smallest enclosing **for** or **while** loop.

```
>>> x = 0
>>> while x < 5:
...     x += 1
...     print x

1
2
3
4
5
```

```
>>> for x in range(0, 100, 5):
...     if x > 10:
...         continue
...     print x

0
5
10
```

```
>>> for x in range(0, 100, 5):
...     print x
...     if x > 15:
...         break

0
5
10
15
20
```

Python scripts

At this point, everyone has graduated from the interpreter! Congratulations are in order.

We will be switching to writing Python files (scripts). Your favorite editor will be just fine.

The extension for Python is **py**.

To run your script, pass it to the python interpreter using the command line.

Command line

```
C:\>python.exe Hello.py
```

Hello World

Let’s try our first Python script, which will print “Hello World” to the user.

- Run program from the terminal.

```
> python HelloWorld.py

Hello World
```

HelloWorld.py

```
#!/usr/bin/env python

print “Hello World”
```

The shebang (#!)

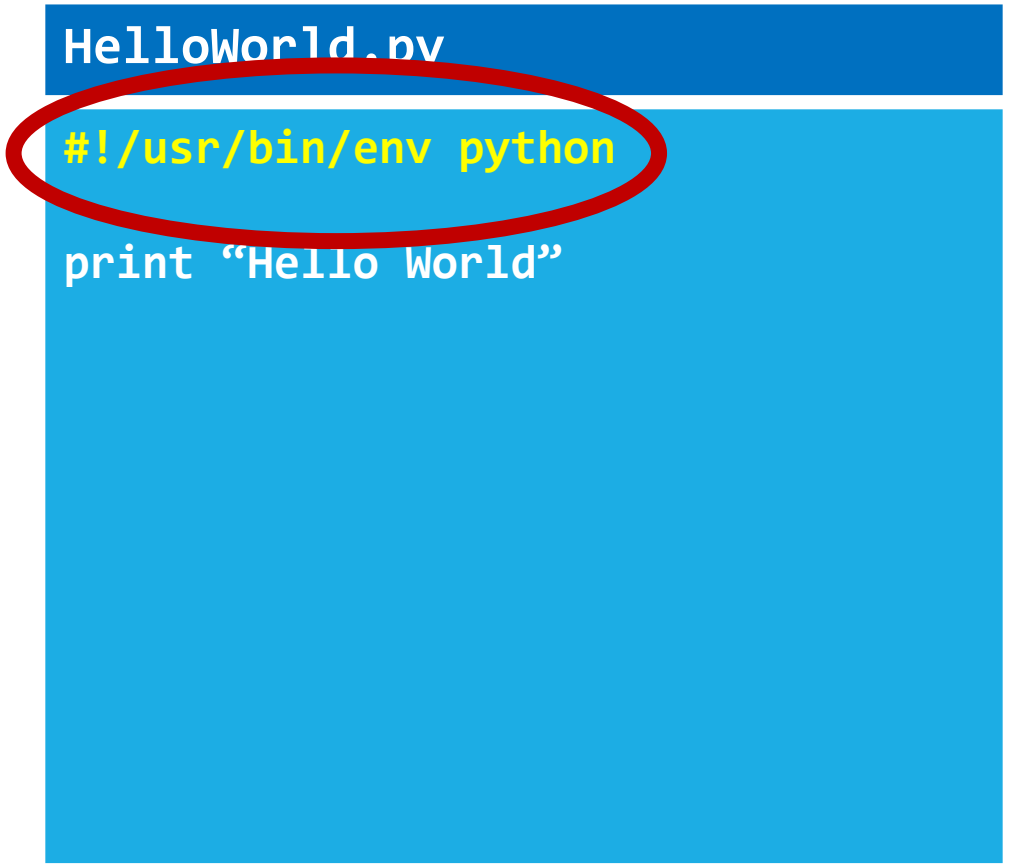
This first line of code is called a shebang.

You can choose to ignore it if you wish!

In Linux, it will allow you to execute the program without having to explicitly tell Linux to call Python.

In Windows you have to do that anyway, so the shebang doesn't help much.

From here on out, we will omit it from our code. Know that you can always add it if you wish.



Functions

Functions

If there's something you want to do more than once, a function will be useful. There are three things to pay attention to when it comes to functions:

- Function definition: this is the name of the function. The function below is named **Func.**
- Input(s): a function may have zero or more inputs. The input to **Func** is **a.**
- Return: a function may return one object to the caller. **Func** returns object **b.**

```
def Func(a):
    b = 2 * a
    return b
```

Functions

Once a function is defined, it can be used by the rest of your program!

The code snippet below calls the function **Func** twice, with inputs **10** and **20**.

Func(**10**) will return **20**. This is assigned to the variable x and printed.

Func(**20**) will return **40**. This is assigned to the variable y and printed.

```
def Func(a):
    b = 2 * a
    return b

x = Func(10)
y = Func(20)
print x
print y
```

Function: example

Our first function is a simple one. It will query the user for a number, and then return the square of that number.

```
> python Squarify.py

enter a number: 50
2500
```

Squarify.py

```
def Square(numIn):
    return numIn ** 2

x = input("enter a number: ")
print Square(x)
```

Function: example

A function need not return anything. In this example, we just need the function to print something to the user.

```
> python Greetings.py

first name: James
last name: Bond
Hello James Bond!
```

Greetings.py

```
def Greeting(F, L):
    print "Hello " + F + " " + L + "!"

first = raw_input("first name: ")
last = raw_input("last name: ")
Greeting(first, last)
```

Function: example

A function may have as many inputs as you like. The Pythag function has two inputs (two sides of a right triangle).

A function could also have no inputs!

```
> python Pythag.py

enter side a: 3
enter side b: 4
side c is: 5.0
```

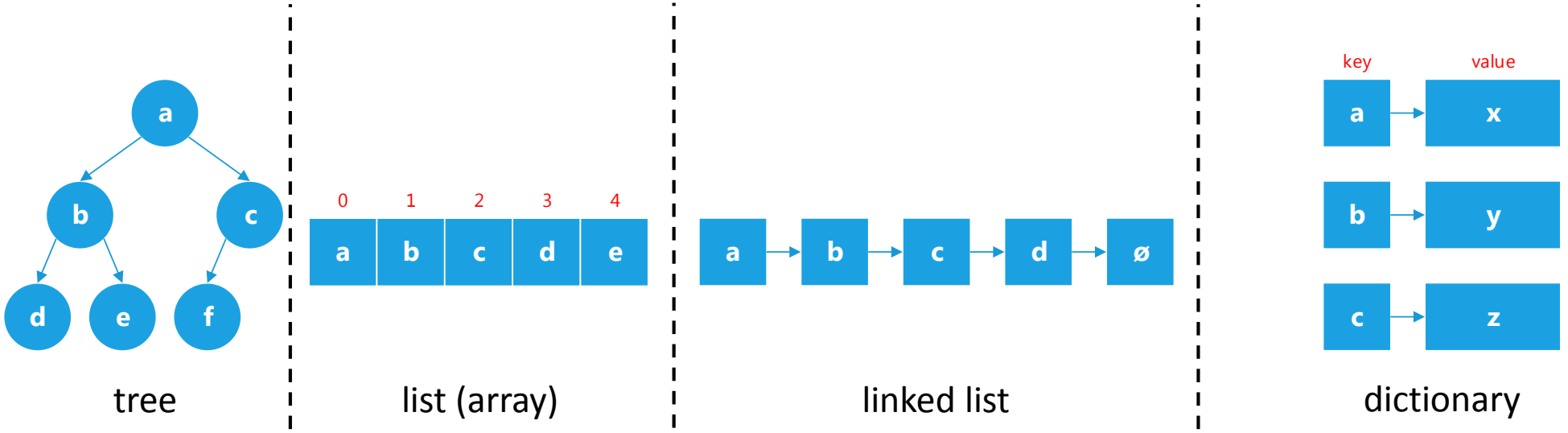
Pythag.py

```
import math

def Pythag(a, b):
    c = math.sqrt(a ** 2 + b ** 2)
    return c

sideA = input("enter side a: ")
sideB = input("enter side b: ")
sideC = Pythag(sideA, sideB)
print "side c is: " + str(sideC)
```

Data Structures

Data Structures

- Data structures are methods of organizing data within a programming language. Each data structure has benefits and drawbacks. Many are beyond the scope of this course. We will be focusing on the python **list** (array) and **dictionary** (hashmap).
- Some examples of data structures are shown below. We will not focus on either tree or linked list, but they are useful to show visually how the data structures differ.



List

The most common data structure in Python is a comma-separated **list** of items. A list is contained within **square brackets**.

Lists can be indexed, assigned or sliced using notation similar to MATLAB (although **zero-based**).

```
Employees = [“Fry”, “Bender”, “Zoidberg”, “Leela”]

testScores = [93.5, 88.3, 91.6]

a = [] # empty list

Teams = [[“Emily”, “Will”], [“Chris”, “Dan”]] # nested

ChessAlive = [pawn0, pawn1, king, queen, knight1]
```

List – accessing and modifying

- Lists can be accessed and modified in python using square brackets.
- Similar to many other programming languages, indexing in python begins with 0.
- Negative indices can be used to access a list beginning with the last element.
- The index used must be within the list. In the below examples, accessing a[5] will return an error.

```
>>> a = [5, 3, -2, 7, 0]
>>> print a[0]
5

>>> print a[2]
-2

>>> print a[-1]
0
```

```
>>> a = [5, 3, -2, 7, 0]
>>> a[0] = 100
>>> print a
[100, 3, -2, 7, 0]

>>> a[-1] = 200
>>> print a
[100, 3, -2, 7, 200]
```

List – slicing

- To access multiple elements simultaneously, the list can be sliced, also using square brackets.
- Slicing can also be used to split a list into two parts, or to make a copy of a list.
- As with the range() function, a[m:n] will contain a[m], a[m+1], …, a[n-1] but **not** a[n]
- If the first argument is omitted, it defaults to 0.
- If the second argument is omitted, it defaults to the length of the list.

```
>>> a = [5, 3, -2, 7, 0]
>>> print a[1:3]
[3, -2]

>>> print a[2:]
[-2, 7, 0]

>>> print a[:2]
[5, 3]
```

```
>>> a = [5, 3, -2, 7, 0]
>>> left = a[:3]
>>> right = a[3:]
>>> print left, right
[5, 3, -2], [7, 0]

>>> print a[:] # make a copy
[5, 3, -2, 7, 0]
```

List – routines

Lists have a number of useful routines. A routine is a function which can be called on any list object. For a complete listing of list's routines, please see this page: https://docs.python.org/2/tutorial/datastructures.html#more-on-lists.

Syntax for calling **routine sort()** on **object x** is: **x.sort()**

For us, it is enough to focus on a few routines:

- append(x) – add a single item to the end of the list.
- extend(x) – add all the elements in list x to the end of the list.
- sort() – sort the list (defaults to ascending order).
- reverse() – reverse the order of the list's elements.

```
>>> a = [5, 3, -2, 7, 0]
>>> a.sort()
>>> print a
[-2, 0, 3, 5, 7]

>>> a.reverse()
>>> print a
[7, 5, 3, 0, -2]
```

List – append()

List's append(x) routine will add x to the end of the list. This gets used quite often.

In the example on the right, we use **append** to keep track of all scores above a certain cutoff.

```
> python HighScores.py

[75, 80, 93]
```

HighScores.py

```
def Filter(s, c):
  hiScores = []
  for score in s:
    if score >= c:
      hiScores.append(score)
  return hiScores

scores = [75, 80, 93, 60, 72, 74]
cutoff = 75
print Filter(scores, cutoff)
```

List – extend()

The extend routine accepts a list as its input. All of the elements in the input list will be added to the end of the called object's list.

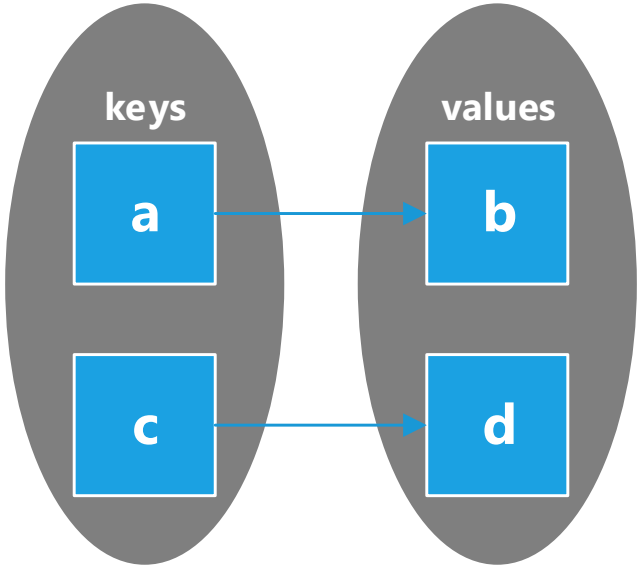
For lists **a** and **b**, **a**.extend(**b**) is equivalent to a concatenation between the lists.

```
>>> a = [1, 3, 5, 7]
>>> b = [2, 4, 6]
>>> a.extend(b)
>>> print a
[1, 3, 5, 7, 2, 4, 6]

>>> b.extend(a)
>>> print b
[2, 4, 6, 1, 3, 5, 7, 2, 4, 6]
```

Dictionary

Also known as a hash table or hash map, the dictionary stores a set of key/value pairs.

Use curly braces "`{ }`" for dictionaries. When you are initializing a dictionary, use comma separated pairs of objects. For example, `{"a":"b", "c":"d"}` will create the dictionary below.



Dictionary – keys and values

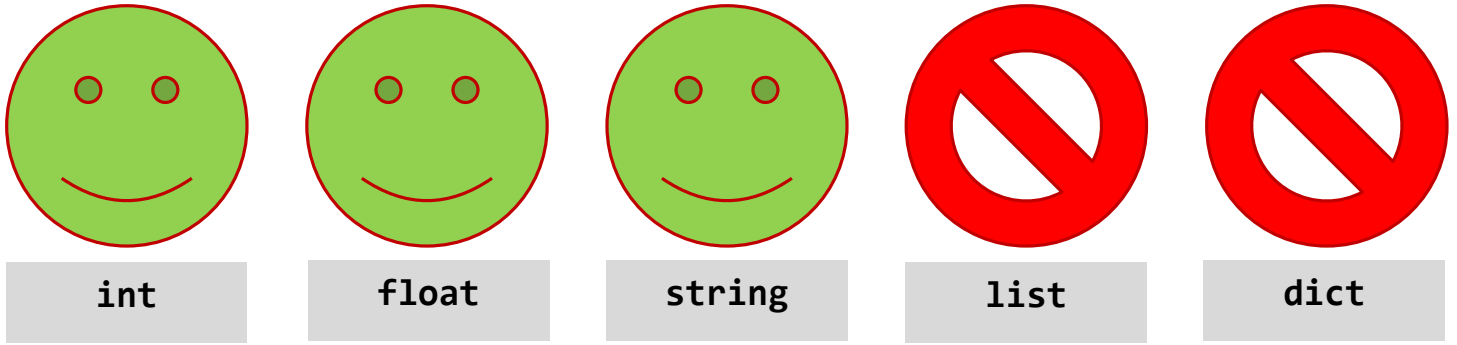
There is a lot of detail on what can be used as a key. It has to do with mutability, and I'll get to it if there's time at the end. If not, there are some backup slides on it. This description on python.org of why strings are immutable is useful to help understanding: https://docs.python.org/2/faq/design.html#why-are-python-strings-immutable.

For this course, we can restrict ourselves to the following:

- The keys may be any immutable type, but for now just know to use: **numbers** and **strings** as keys.
- Mutable types may not be used as keys. This means the key cannot be a **list** or **dictionary**!

Values can be **any type**.

Types allowed as key:



Dictionary – interaction

Interacting with a **dictionary** is very similar to interacting with a **list**.

With a list, the **index** had to be an **integer**, starting with 0.

In a **dictionary**, the index is the **key**, and the key can be any of the types we just discussed.

The **value** can be whatever we want. It can even be a list or another dictionary!

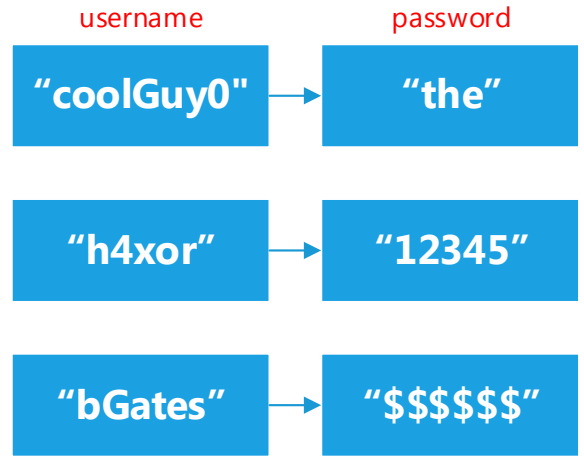
```
>>> d = {"a":27, "b":"dog"}
>>> print d
{'a': 27, 'b': 'dog'}

>>> d["c"] = [1, 3] # add key "c"
>>> print d
{'a': 27, 'c': [1, 3], 'b': 'dog'}
```

```
>>> print len(d)
3

>>> del d["a"] # remove key "a"
>>> print d, len(d)
{'c': [1, 3], 'b': 'dog'} 2
```

Dictionary – example

Using a dictionary is a really easy way to deal with objects and their properties. In this example, we are storing unencrypted passwords for three users.

```
>>> pwords = {"coolGuy0" : "the", "h4xor" : "12345",
"bGates" : "$$$$$$"} # create a dictionary called pwords

>>> print pwords # print the dictionary
{'bGates': '$$$$$$', 'coolGuy0': 'the', 'h4xor': '12345'}

>>> print pwords["bGates"] # print a value
$$$$$$

>>> pwords["h4xor"] = "54321“ # change a value

>>> print pwords # print the new dictionary
{'bGates': '$$$$$$', 'coolGuy0': 'the', 'h4xor': '54321'}
```



Dictionary – example 1

A dictionary is a great way to count unique elements.

In this example we are keeping track of how many times each grade was observed.

```
> python Repeats.py

A: 2
A+: 1
B: 2
C: 1
```

Repeats.py

```
def Repeats(grades):
    D = {}
    for g in grades:
        if g in D:
            D[g] += 1
        else:
            D[g] = 1
   return D

grades = ["A", "A+", "B", "B", "A", "C"]
rep = Repeats(grades)
for grade in rep:
    print grade + ": " + str(rep[grade])
```

Dictionary – example 2

Keep track of user's ages

```
> python AgeList.py

enter name: bob
enter age: 30

enter name: bob
enter age: 35

enter name: jill
enter age: 32

bob->[30, 35]
jill->[32]
```

AgeList.py

```
D = {}
for person in range(3):
    name = raw_input("enter name: ")
    age = int(input("enter age: "))
    if name in D:
        D[name].append(age)
    else:
        D[name] = [age]
    print ""

for person in D:
    print d + "->" + str(D[d])
```

Dictionary – example 3

A very simple game involving the player and an enemy.

Dictionary is used to keep track of each character's attributes.

```
> python Game.py

player wins!
```

Game.py

```
# set up characters
player = {'health': 50, 'damage': 300, 'armor': []}
enemy = {'health': 250, 'damage': 200, 'armor':
['Laser Armor']}

# consume armor
for armor in enemy['armor']:
    enemy['health'] += 25
for armor in player['armor']:
    player['health'] += 25

# fight!
if player['damage'] > enemy['health']:
    print "player wins!"
elif enemy['damage'] > player['health']:
    print "player loses!"
else:
    print "stalemate!"
```

Importing libraries

Libraries

Widespread Python support means a rich selection of third-party libraries for multiple platforms.

This means you can spend less time reinventing the wheel.

Import a library into your script using the **import** keyword.

```
>>> import math
>>> p = math.pi
>>> r = 10
>>> print p * math.pow(r, 2)
314.159265359

>>> print 2 * p * r
62.8318530718
```

```
>>> import random
>>> print random.randrange(10)
4

>>> print random.randrange(10)
9

>>> print random.randrange(10)
5
```

Python standard library

There are a large number of libraries that are included by default in your Python distribution. These are collectively known as the Python standard library. There are many, and of course you don't have to memorize them all. If you're curious, see here: https://docs.python.org/2/library/.

We just saw two of them! **math** and **random** are both included in the Python standard library.

Since they are included by default, you do not need to **install** these libraries.

Practical Example 1

IMAGE PROCESSING

Image processing

We can take advantage of the **Pillow** library to make image processing a breeze.

This short program will let us smooth out an image a specified amount.

```
> python MedianFilterA.py

enter smoothness: 5
processing: 1 of 5
processing: 2 of 5
processing: 3 of 5
processing: 4 of 5
processing: 5 of 5
finished!
```

MedianFilterA.py

```
from PIL import Image
from PIL import ImageFilter

im = Image.open(“nature.jpg") # load image
s = int(raw_input("enter smoothness: "))
if s < 0 or s > 10: # make sure input is reasonable
    print "smoothness must be in range [0, 10]"
    exit()

for i in range(s):
    print "processing: " + str(i + 1) + " of " + str(s)
    im = im.filter(ImageFilter.MedianFilter(5))
print "finished"

imOutName = “nature_" + str(s) + ".jpg"
im.save(imOutName, "JPEG")
im.show()
```

Image processing

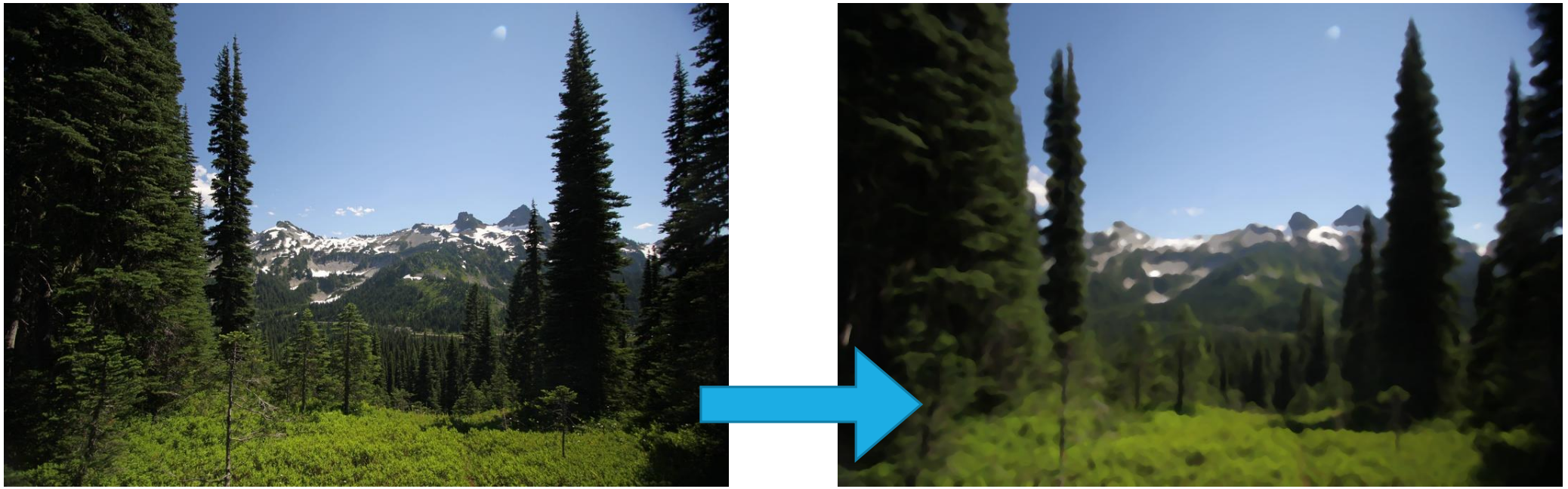


Image processing

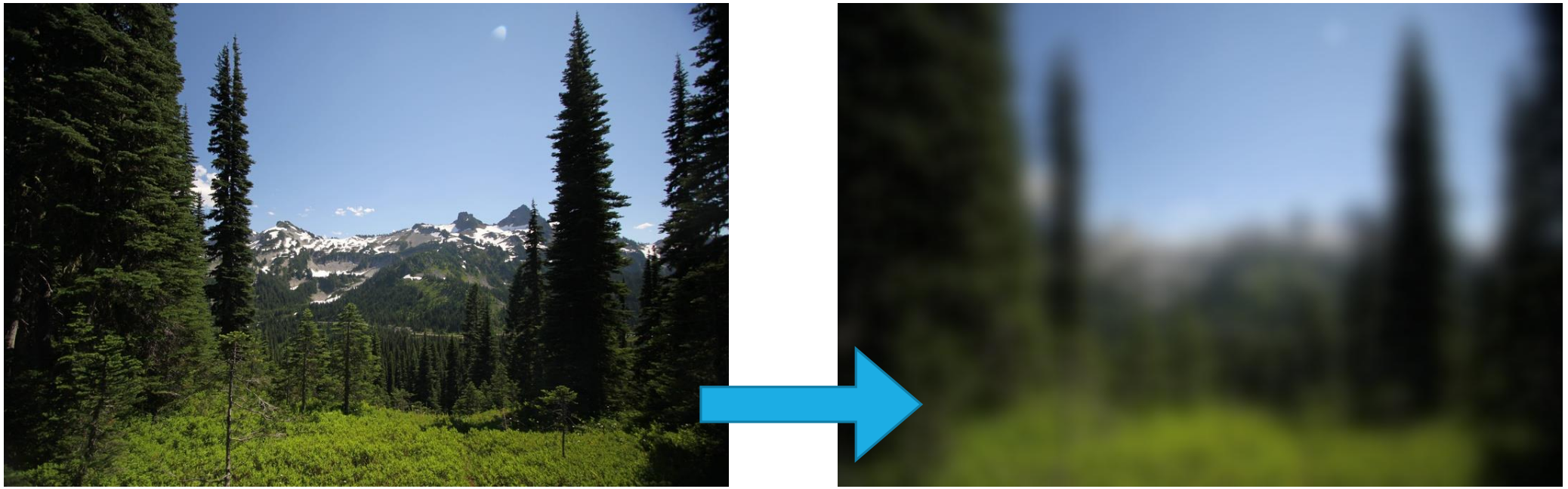
We can easily modify our program to add a new feature. Now we let the user select between two different filters.

```
> python MedianFilterB.py

enter smoothness: 5
select filter: 1
processing: 1 of 5
processing: 2 of 5
processing: 3 of 5
processing: 4 of 5
processing: 5 of 5
finished!
```

MedianFilterB.py

```
from PIL import Image
from PIL import ImageFilter

im = Image.open("nature.jpg") # load image
s = int(raw_input("enter smoothness: "))
r = int(raw_input("select filter: "))
if (s < 0 or s > 10) or (r < 0 or r > 1):
    print "smoothness must be in range [0, 10]"
    print "filter should be in range [0, 1]"
    exit()

for i in range(s):
    print "processing: " + str(i + 1) + " of " + str(s)
    if r == 0:
        im = im.filter(ImageFilter.MedianFilter(5))
    elif r == 1:
        im = im.filter(ImageFilter.GaussianBlur(5))
print "finished!"

imOutName = "nature" + str(s) + "_" + str(r) + ".jpg"
im.save(imOutName, "JPEG")
im.show()
```

Image processing



Practical Example 2

ANAGRAM DETECTION

Anagram detection

This is a common interview question, and it’s really easy to solve with a few key ideas.

Once we have the ideas, it can be done in surprisingly little code.

Goal: given a word in English, find all of its anagrams.

Anagram detection

This is a common interview question, and it's really easy to solve with a few key ideas.

Once we have the ideas, it can be done in surprisingly little code.

Goal: given a word in English, find all of its anagrams.

Hint: the **dictionary** data structure will help!

We can solve this in two steps:

- Step one: process a wordlist using our key idea.
- Step two: pick a word and list all its anagrams.

Anagram detection

Key idea:

- We need a representation where any two words which are anagrams point to the same thing.
- Two words which are anagrams have exactly the same letters, but in different order.
- Think about sorting…

Anagram detection

Key idea:

- We need a representation where any two words which are anagrams point to the same thing.
- Two words which are anagrams have exactly the same letters, but in different order.
- Think about sorting…

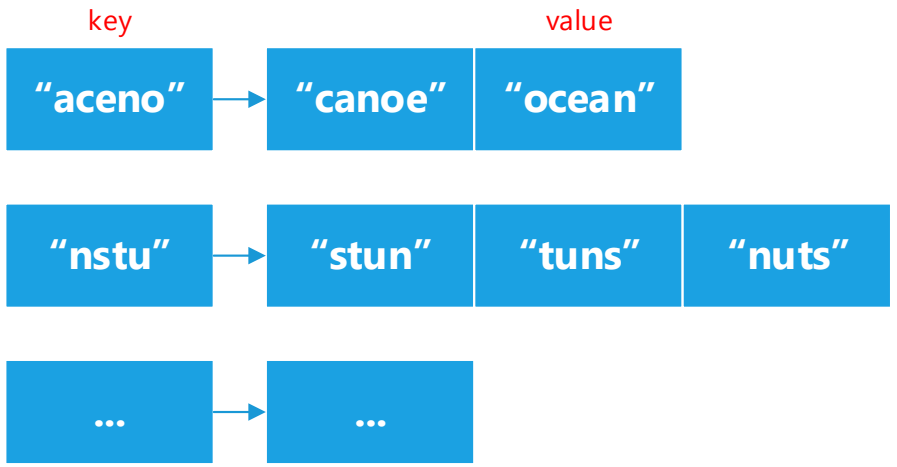
Anagram detection

Key idea:

- We need a representation where any two words which are anagrams point to the same thing.
- Two words which are anagrams have exactly the same letters, but in different order.
- Think about sorting…

Now we can use a **dictionary**, where the **key** is the sorted string, and the **value** is a list of all words which have that sorted string.

Once we have built up this dictionary, using a wordlist, it's quick and easy to find all the anagrams for any given word!



Anagram detection

We have the idea, now let's put it into code!

The function GenKey converts the input string to lowercase and then sorts it.

```
> python Anagram.py

processing word list...
done
enter word: horse
['heros', 'horse', 'osher', 'shore']
```

Anagram.py

```
def GenKey(stringIn):
    stringIn = stringIn.lower() # convert to lowercase
    a = list(stringIn) # convert string to list
    a.sort() # sort list
    return "".join(a) # convert list to string, return

D = {}
print "processing word list..."
with open("wlist.txt") as f:
    for line in f:
        word = line.strip() # remove whitespace, newlines
        k = GenKey(word) # k is the sorted string
        if k in D:
            D[k].append(word)
        else:
            D[k] = [word]
print "done"

word = raw_input("enter word: ")
print D[GenKey(word)]
```

Thanks!

Acknowledgments

This class took a lot of time and effort to put together. I didn't do it alone. I really appreciate the help provided by the following people:

- Prof. Truong Nguyen (UCSD ECE Department Chair, Professor)
- San Diego Python Users Group
- Trey Hunner (---)
- Dr. Sayanan Sivaraman (Apple, UCSD PhD 2013)
- Dr. Karl Ni (Lawrence Livermore National Laboratory, UCSD PhD 2008)
  - Please see Karl's upcoming course "**Build Your First AI Algorithm**" using **Python** and the **Keras** toolbox.