Introduction Python Installation

**Workﬂow/ Setup Examples**

Python Basics

Autograding

Question 1: Addition

Question 2: buyLotsOfFruit function

Question 3: shopSmart function

Submission

Python Basics

# Required Files

You can download all of the ﬁles associated with the Python mini-tutorial as a zip archive: [python\_basics.zip](https://inst.eecs.berkeley.edu/~cs188/su22/assets/files/python_basics.zip)

If you are rusty or don't have experience with Python, you are encouraged to do this exercise. Otherwise, you can navigate using the Table of Contents to look at the things you want to read about, although the entire section would still be a good review.

# Table of Contents

Invoking the Interpreter Operators

Strings

Dir and Help

Built-in Data Structures Lists

Tuples Sets

Dictionaries Writing Scripts Indentation

Tabs vs Spaces Writing Functions Object Basics

Deﬁning Classes Using Objects

Static vs Instance Variables Tips and Tricks

Troubleshooting More References

The programming assignments in this course will be written in [Python](http://www.python.org/about/), an interpreted, object-oriented language that shares some features with both Java and Scheme. This tutorial will walk through the primary syntactic constructions in Python, using short examples.

We encourage you to type all python shown in the tutorial onto your own machine. Make sure it responds the same way.

You may ﬁnd the Troubleshooting section helpful if you run into problems. It contains a list of the frequent problems students have encountered when following this tutorial.

# Invoking the Interpreter

Python can be run in one of two modes. It can either be used *interactively*, via an interpeter, or it can be called from the command line to execute a *script*. We will ﬁrst use the Python interpreter interactively.

You invoke the interpreter using the command at the command

python

Prompt. You can get info on python using command

python -i

.

**Copy**

C:\Users\user\Dropbox\teaching\CMPT310\Projects>python -i

Python 3.10.9 | packaged by Anaconda, Inc. | (main, Mar 1 2023, 18:18:15) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> quit()

# Operators

The Python interpreter can be used to evaluate expressions, for example simple

arithmetic expressions. If you enter such expressions at the prompt ( be evaluated and the result will be returned on the next line.

>>>

**Copy**

>>> 1 + 1

2

>>> 2 \* 3

6

Boolean operators also exist in Python to manipulate primitive values.

True

False

) they will

and

**Copy**

>>> 1 == 0

False

>>> not (1 == 0)

True

>>> (2 == 2) and (2 == 3)

False

>>> (2 == 2) or (2 == 3)

True

# Strings

Like Java, Python has a built-in string type. The string concatenation on string values.

operator is overloaded to do



+

**Copy**

>>> 'artificial' + "intelligence"

'artificialintelligence'

There are many built-in methods which allow you to manipulate strings.

**Copy**

>>> 'artificial'.upper() 'ARTIFICIAL'

>>> 'HELP'.lower()

'help'

>>> len('Help') 4

Notice that we can use either single quotes string. This allows for easy nesting of strings.

' '

We can also store expressions into variables.

or double quotes

to surround

**Copy**

>>> s = 'hello world'

>>> print(s) hello world

>>> s.upper()

'HELLO WORLD'

>>> len(s.upper()) 11

>>> num = 8.0

>>> num += 2.5

>>> print(num)

10.5

In Python, you do not have to declare variables before you assign to them.

# Exercise: Dir and Help

Learn about the methods Python provides for strings. To see what methods Python

dir

help

provides for a datatype, use the

" "

and

commands:

>>> s = 'abc'

**Copy**

>>> dir(s)

[' add ', ' class ', ' contains ', ' delattr ', ' doc ', ' eq ', ' ge ', ' getattribute ',

' getitem ', ' getnewargs ', ' getslice ', ' gt ',

' hash ', ' init ', ' le ', ' len ', ' lt ',

' mod ', ' mul ', ' ne ', ' new ', ' reduce ', ' reduce\_ex ', ' repr ', ' rmod ', ' rmul ',

' setattr ', ' str ', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find',

'index', 'isalnum', 'isalpha', 'isdigit', 'islower',

'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',

'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> help(s.find)

Help on built-in function find:

find(...) method of builtins.str instance S.find(sub[, start[, end]]) -> int

Return the lowest index in S where substring sub is found,

such that sub is contained within S[start:end].

Optional

arguments start and end are interpreted as in slice notation.

Return -1 on failure.

>>> s.find('b') 1

Try out some of the string functions listed in around the method name).

dir

(ignore those with underscores ‘\_’

Note: Ignore functions with underscores “\_” around the names; these are private helper methods. Press ‘q’ to back out of a help screen.

# Built-in Data Structures

Python comes equipped with some useful built-in data structures, broadly similar to Java’s collections package.

## Lists

*Lists* store a sequence of mutable items:

**Copy**

>>> fruits = ['apple', 'orange', 'pear', 'banana']

>>> fruits[0] 'apple'

We can use the operator to do list concatenation:



+

**Copy**

>>> otherFruits = ['kiwi', 'strawberry']

>>> fruits + otherFruits

>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']

Python also allows negative-indexing from the back of the list. For instance,

will access the last element :

fruits[-1]

'banana'

**Copy**

>>> fruits[-2] 'pear'

>>> fruits.pop() 'banana'

>>> fruits

['apple', 'orange', 'pear']

>>> fruits.append('grapefruit')

>>> fruits

['apple', 'orange', 'pear', 'grapefruit']

>>> fruits[-1] = 'pineapple'

>>> fruits

['apple', 'orange', 'pear', 'pineapple']

We can also index multiple adjacent elements using the slice operator. For instance,

, returns a list containing the elements at position 1 and 2. In general will get the elements in

fruits[1:3]

fruits[start:stop]

start, start+1, ..., stop-1

. We can also do which

fruits[start:]

start

fruits[:end]

returns all elements starting from the

index. Also

will

return all elements before the element at position :

end

**Copy**

>>> fruits[0:2] ['apple', 'orange']

>>> fruits[:3]

['apple', 'orange', 'pear']

>>> fruits[2:] ['pear', 'pineapple']

>>> len(fruits)

4

The items stored in lists can be any Python data type. So for instance we can have lists of lists:

**Copy**

>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]

>>> lstOfLsts[1][2] 3

>>> lstOfLsts[0].pop() 'c'

>>> lstOfLsts

[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]

# Exercise: Lists

Play with some of the list functions. You can ﬁnd the methods you can call on an

dir

help

object via the

and get information about them via the

command:

**Copy**

>>> dir(list)

[' add ', ' class ', ' contains ', ' delattr ', ' delitem ',

' delslice ', ' doc ', ' eq ', ' ge ', ' getattribute ',

' getitem ', ' getslice ', ' gt ', ' hash ',

' iadd ', ' imul ',

' init ', ' iter ', ' le ', ' len ', ' lt ', ' mul ', ' ne ',

' new ', ' reduce ', ' reduce\_ex ', ' repr ', ' reversed ',

' rmul ', ' setattr ', ' setitem ', ' setslice ', ' str ',

'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',

'sort']

**Copy**

>>> help(list.reverse)

Help on built-in function reverse:

reverse(...)

L.reverse() -- reverse \\*IN PLACE\\*

**Copy**

>>> lst = ['a', 'b', 'c']

>>> lst.reverse()

>>> ['c', 'b', 'a']`

## Tuples

A data structure similar to the list is the *tuple*, which is like a list except that it is immutable once it is created (i.e. you cannot change its content once created). Note that tuples are surrounded with parentheses while lists have square brackets.

**Copy**

>>> pair = (3, 5)

>>> pair[0] 3

>>> x, y = pair

>>> x 3

>>> y

5

>>> pair[1] = 6

TypeError: object does not support item assignment

The attempt to modify an immutable structure raised an exception. Exceptions indicate errors: index out of bounds errors, type errors, and so on will all report exceptions in this way.

## Sets

A *set* is another data structure that serves as an unordered list with no duplicate items. Below, we show how to create a set:

**Copy**

>>> shapes = ['circle', 'square', 'triangle', 'circle']

>>> setOfShapes = set(shapes)

Another way of creating a set is shown below:

>>> setOfShapes = {‘circle’, ‘square’, ‘triangle’, ‘circle’}

Next, we show how to add things to the set, test if an item is in the set, and perform common set operations (difference, intersection, union):

**Copy**

>>> setOfShapes

set(['circle', 'square', 'triangle'])

>>> setOfShapes.add('polygon')

>>> setOfShapes

set(['circle', 'square', 'triangle', 'polygon'])

>>> 'circle' in setOfShapes True

>>> 'rhombus' in setOfShapes False

>>> favoriteShapes = ['circle', 'triangle', 'hexagon']

>>> setOfFavoriteShapes = set(favoriteShapes)

>>> setOfShapes - setOfFavoriteShapes set(['square', 'polygon'])

>>> setOfShapes & setOfFavoriteShapes

set(['circle', 'triangle'])

>>> setOfShapes | setOfFavoriteShapes

set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])

**Note that the objects in the set are unordered; you cannot assume that their traversal or print order will be the same across machines!**

## Dictionaries

The last built-in data structure is the *dictionary* which stores a map from one type of object (the key) to another (the value). The key must be an immutable type (string, number, or tuple). The value can be any Python data type.

Note: In the example below, the printed order of the keys returned by Python could be different than shown below. The reason is that unlike lists which have a ﬁxed ordering, a dictionary is simply a hash table for which there is no ﬁxed ordering of the keys (like HashMaps in Java). The order of the keys depends on how exactly the hashing algorithm maps keys to buckets, and will usually seem arbitrary. Your code should not rely on key ordering, and you should not be surprised if even a small modiﬁcation to how your code uses a dictionary results in a new key ordering.

**Copy**

>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}

>>> studentIds['turing'] 56.0

>>> studentIds['nash'] = 'ninety-two'

>>> studentIds

{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}

>>> del studentIds['knuth']

>>> studentIds

{'turing': 56.0, 'nash': 'ninety-two'}

>>> studentIds['knuth'] = [42.0, 'forty-two']

>>> studentIds

{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}

>>> studentIds.keys() ['knuth', 'turing', 'nash']

>>> studentIds.values()

[[42.0, 'forty-two'], 56.0, 'ninety-two']

>>> studentIds.items()

[('knuth', [42.0, 'forty-two']), ('turing',56.0), ('nash', 'ninety-two')]

>>> len(studentIds) 3

As with nested lists, you can also create dictionaries of dictionaries.

# Exercise: Dictionaries

help

Use

dir

and

to learn about the functions you can call on dictionaries.

# Writing Scripts

Now that you’ve got a handle on using Python interactively, let’s write a simple

Python script that demonstrates Python’s loop. Open the ﬁle called

for

, which should contain the following code:

foreach.py

**Copy**

# This is what a comment looks like

fruits = ['apples', 'oranges', 'pears', 'bananas'] for fruit in fruits:

print(fruit + ' for sale')

fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}

for fruit, price in fruitPrices.items(): if price < 2.00:

print('%s cost %f a pound' % (fruit, price)) else:

print(fruit + ' are too expensive!')

At the command line, use the following command in the directory containing

:

foreach.py

**Copy**

C:\Users\user\Dropbox\teaching\CMPT310\session0\tutorial\python\_basics>python foreach.py

apples for sale

oranges for sale

pears for sale

bananas for sale

apples are too expensive!

oranges cost 1.500000 a pound

pears cost 1.750000 a pound

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial; that’s due to the fact that we’re looping over dictionary keys, which are unordered. To learn more about control structures (e.g.,

[else](https://docs.python.org/3.6/tutorial/)

[and topic.](https://docs.python.org/3.9/tutorial/)

[if](https://docs.python.org/3.6/tutorial/)

[) in Python, check out the ofﬁcial Python tutorial section on this](https://docs.python.org/3.9/tutorial/)

If you like functional programming, you might also like and :

map

filter

**Copy**

if 0 == 1:

print('We are in a world of arithmetic pain') print('Thank you for playing')

**Copy**

>>> list(map(lambda x: x \* x, [1, 2, 3]))

[1, 4, 9]

>>> list(filter(lambda x: x > 3, [1, 2, 3, 4, 5, 4, 3, 2,

1]))

[4, 5, 4]

The next snippet of code demonstrates Python’s *list comprehension* construction:

**Copy**

print(oddNumsPlusOne)

|  |  |  |
| --- | --- | --- |
| nums = [1, 2, 3, | 4, 5, 6] |  |
| plusOneNums = [x | + 1 for x in | nums] |
| oddNums = [x for | x in nums if | x % 2 == 1] |
| print(oddNums)  oddNumsPlusOne = | [x + 1 for x | in nums if x % 2 == 1] |

This code is in a ﬁle called , which you can run:

listcomp.py

**Copy**

$ python listcomp.py [1, 3, 5]

[2, 4, 6]

# Exercise: List Comprehensions

Write a list comprehension which, from a list, generates a lowercased version of each string that has length greater than ﬁve. You can ﬁnd the solution in

.

listcomp2.py

# Beware of Indendation!

Unlike many other languages, Python uses the indentation in the source code for interpretation. So for instance, for the following script:

will output:

Thank you for playing

But if we had written the script as

**Copy**

if 0 == 1:

print('We are in a world of arithmetic pain') print('Thank you for playing')

there would be no output. The moral of the story: be careful how you indent! It’s best to use four spaces for indentation – that’s what the course code uses.

# Tabs vs Spaces

Because Python uses indentation for code evaluation, it needs to keep track of the level of indentation across code blocks. This means that if your Python ﬁle switches from using tabs as indentation to spaces as indentation, the Python interpreter will not be able to resolve the ambiguity of the indentation level and throw an exception. Even though the code can be lined up visually in your text editor, Python “sees” a change in indentation and most likely will throw an exception (or rarely, produce unexpected behavior).

This most commonly happens when opening up a Python ﬁle that uses an indentation scheme that is opposite from what your text editor uses (aka, your text editor uses spaces and the ﬁle uses tabs). When you write new lines in a code block, there will be a mix of tabs and spaces, even though the whitespace is aligned. For a longer discussion on tabs vs spaces, see [this](http://stackoverflow.com/questions/119562/tabs-versus-spaces-in-python-programming) discussion on StackOverﬂow.

# Writing Functions

As in Java, in Python you can deﬁne your own functions:

**Copy**

fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears':

1.75}

def buyFruit(fruit, numPounds): if fruit not in fruitPrices:

print("Sorry we don't have %s" % (fruit)) else:

cost = fruitPrices[fruit] \* numPounds

print("That'll be %f please" % (cost))

# Main Function

if name == ' main ': buyFruit('apples', 2.4)

buyFruit('coconuts', 2)

Rather than having a function as in Java, the

main

name == ' main '

check is used to delimit expressions which are executed when the ﬁle is called as a script from the command line. The code after the main check is thus the same sort of

code you would put in a function in Java.

main

Save this script as *fruit.py* and run it:

**Copy**

$ python fruit.py That'll be 4.800000 please

Sorry we don't have coconuts

# Advanced Exercise

Write a function in Python using list comprehensions. Use the ﬁrst

quickSort

element as the pivot. You can ﬁnd the solution in .

quickSort.py

# Object Basics

Although this isn’t a class in object-oriented programming, you’ll have to use some objects in the programming projects, and so it’s worth covering the basics of objects in Python. An object encapsulates data and provides functions for interacting with that data.

## Deﬁning Classes

Here’s an example of deﬁning a class named :

FruitShop

class FruitShop:

**Copy**

def

init (self, name, fruitPrices): """

name: Name of the fruit shop

1.75}

"""

fruitPrices: Dictionary with keys as fruit strings and prices for values e.g.

{'apples': 2.00, 'oranges': 1.50, 'pears':

self.fruitPrices = fruitPrices self.name = name

print('Welcome to %s fruit shop' % (name))

def getCostPerPound(self, fruit): """

fruit: Fruit string

Returns cost of 'fruit', assuming 'fruit' is in our inventory or None otherwise

"""

if fruit not in self.fruitPrices: return None

return self.fruitPrices[fruit]

def getPriceOfOrder(self, orderList): """

orderList: List of (fruit, numPounds) tuples

Returns cost of orderList, only including the values of

fruits that this fruit shop has.

"""

totalCost = 0.0

for fruit, numPounds in orderList: costPerPound = self.getCostPerPound(fruit) if costPerPound != None:

totalCost += numPounds \* costPerPound return totalCost

def getName(self): return self.name

The class has some data, the name of the shop and the prices per

FruitShop

pound of some fruit, and it provides functions, or methods, on this data. What advantage is there to wrapping this data in a class?

1. Encapsulating the data prevents it from being altered or used inappropriately,
2. The abstraction that objects provide make it easier to write general-purpose code.

## Using Objects

So how do we make an object and use it? Make sure you have the

FruitShop

implementation in . We then import the code from this ﬁle (making it

shop.py

import shop

shop.py

accessible to other scripts) using ﬁle. Then, we can create

FruitShop

, since objects as follows:

is the name of the

**Copy**

import shop

shopName = 'the Berkeley Bowl'

fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}

berkeleyShop = shop.FruitShop(shopName, fruitPrices)

applePrice = berkeleyShop.getCostPerPound('apples') print(applePrice)

print('Apples cost $%.2f at %s.' % (applePrice, shopName))

otherName = 'the Stanford Mall'

otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50,

'peaches': 8.75}

otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)

otherPrice = otherFruitShop.getCostPerPound('apples') print(otherPrice)

print('Apples cost $%.2f at %s.' % (otherPrice, otherName))

print("My, that's expensive!")

This code is in ; you can run it like this:

shopTest.py

**Copy**

$ python shopTest.py Welcome to the Berkeley Bowl fruit shop 1.0

Apples cost $1.00 at the Berkeley Bowl. Welcome to the Stanford Mall fruit shop 4.5

Apples cost $4.50 at the Stanford Mall.

My, that's expensive!

So what just happended? The the functions and classes in

shop.py

berkeleyShop = shop.FruitShop(shopName, fruitPrices)

statement told Python to load all of

. The line

import shop

*instance* of the

class deﬁned in *shop.py*, by calling the

FruitShop

constructs an

function in that class. Note that we only passed two arguments in, while

init

init

seems to take three arguments:

(self, name, fruitPrices)

this is that all methods in a class have

self

. The reason for as the ﬁrst argument. The

variable’s value is automatically set to the object itself; when calling a method, you

self

only supply the remaining arguments. The variable contains all the data

self

fruitPrices

this

(

name

and

) for the current speciﬁc instance (similar to in

[Java). The print statements use the substitution operator (described in the Python docs if you’re curious).](https://docs.python.org/2/library/stdtypes.html#string-formatting)

## Static vs Instance Variables

The following example illustrates how to use static and instance variables in Python.

Create the containing the following code:

person\_class.py

**Copy**

class Person:

population = 0

def init (self, myAge):

self.age = myAge Person.population += 1

def get\_population(self):

return Person.population

def get\_age(self):

return self.age

We ﬁrst compile the script:

$ python person\_class.py

Now use the class as follows:

**Copy**

>>> import person\_class

>>> p1 = person\_class.Person(12)

>>> p1.get\_population() 1

>>> p2 = person\_class.Person(63)

>>> p1.get\_population() 2

>>> p2.get\_population() 2

>>> p1.get\_age() 12

>>> p2.get\_age()

63

In the code above,

age

population

is an instance variable and

is a static variable.

has its own

Person

age

is shared by all instances of the variable.

class whereas each instance

population

**Copy**

>>> reload(shop)

# More Python Tips and Tricks

This tutorial has brieﬂy touched on some major aspects of Python that will be relevant to the course. Here are some more useful tidbits:

Use to generate a sequence of integers, useful for generating

range

traditional indexed loops:

for

**Copy**

for index in range(3):

print(lst[index])

After importing a ﬁle, if you edit a source ﬁle, the changes will not be immediately propagated in the interpreter. For this, use the command:

reload

# Troubleshooting

These are some problems (and their solutions) that new Python learners commonly encounter.

**Problem:** ImportError: No module named py

**Solution:** For import statements with include the ﬁle extension (i.e. the

.py

NOT:

import shop

import shop.py

, do *not*

string). For example, you should use:

import <package-name>

**Problem:** NameError: name ‘MY VARIABLE’ is not deﬁned Even after importing you may see this.

**Solution:** To access a member of a module, you have to type

, where is the name of the

MODULE NAME.MEMBER NAME

MODULE NAME

.py

ﬁle, and to access.

MEMBER NAME

is the name of the variable (or function) you are trying

**Problem:** TypeError: ‘dict’ object is not callable

**Solution:** Dictionary looks up are done using square brackets: [ and ]. NOT parenthesis: ( and ).

**Problem:** ValueError: too many values to unpack

**Solution:** Make sure the number of variables you are assigning in a loop

for

matches the number of elements in each item of the list. Similarly for working with tuples.

For example, if

is a tuple of two elements (e.g.

) then the following code would cause the “too

pair

pair =('apple', 2.0)

many values to unpack error”:

(a, b, c) = pair

Here is a problematic scenario involving a loop:

for

**Copy**

pairList = [('apples', 2.00), ('oranges', 1.50),

('pears', 1.75)]

for fruit, price, color in pairList:

print('%s fruit costs %f and is the color %s' %

(fruit, price, color))

**Problem:** AttributeError: ‘list’ object has no attribute ‘length’ (or something similar)

**Solution:** Finding length of lists is done using .

len(NAME OF LIST)

**Problem:** Changes to a ﬁle are not taking effect.

**Solution:**

1. Make sure you are saving all your ﬁles after any changes.
2. If you are editing a ﬁle in a window different from the one you are using to

execute python, make sure you to guarantee

reload(\_YOUR\_MODULE\_)

your changes are being reﬂected. works similarly to .

reload

import

# More References

The place to go for more Python information: [www.python.org](http://www.python.org/)