Python3 -m pip install <package-name>

Python –m pip install <package-name>

open(name[, mode[, buffering]]) 例子

open (‘egg.csv’, ‘rb’) rb r = read b = binary mode for improve portability

import csv

>>> with open('eggs.csv', 'rb') as csvfile:

... spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')

... for row in spamreader:

... print ', '.join(row)

**Python**

**Mutable - list**

Immutable- string, number, tuple

**自己总结的 编程几项重要常识 牢记:**

1. 封装. Encapsulation）是指，一种将抽象性函式接口的实作细节部份包装、隐藏起来的方法。同时，它也是一种防止外界呼叫端，去存取物件内部实作细节的手段，这个手段是由编程语言本身来提供的。这两个概念有一些不同，但通常被混合使用。封装被视为是面向对象的四项原则之一. C 语言实例:

namespace Encapsulation

{

class Program

{

public class Account

{

**private** decimal accountBalance = 500.00m;

public decimal CheckBalance()

{

return accountBalance;

}

}

static void Main()

{

Account myAccount = new Account();

decimal myBalance = myAccount.CheckBalance();

**/\* Main方法能够通过public的“CheckBalance”方法确认账户余额，但是不能更改它 . 通过 private 实现\*/**

}

}

}

**CODING Note -1 :**

在python中 None, False, 空字符串"", 0, 空列表[], 空字典{}, 空元组()都相当于False ，即：

**not None == not False == not '' == not 0 == not [] == not {} == not ()**

**if x:**

**即 if x not None , not False , not '' ,not 0 ,not [] ,not {} , not ()**

**CODING NOTE-2:**

filter\_count = {

word: input\_count[word] for word in input\_count

if len(word) >= min\_length}

combine of for loop, if , create dictionary

1.1. Why Python?

A programming language is the tool we use to construct a sequence of instructions that will tell the computer what we want it to do.

There are hundreds of programming languages in the world. There is no best language. Different languages are better or worse for different kinds of applications.

In this course, we will learn Python.

* It is a fun programming language named after the comedy series Monty Python.
* It is an **interpreted** language, created in the 90s.
* It is programming at its **simplest**.
* It is **friendly**, yet powerful.
* Python is a relatively recent addition to the universe of languages, and is still growing in popularity.

In this course you will also learn how to write programs that solve problems. This skill can be transferred to many languages.

The core philosophy of Python is summarized by the document "PEP 20 (The Zen of Python)"

The Zen of Python by Tim Peters – Excerpts

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

优美胜于丑陋（Python 以编写优美的代码为目标）

明了胜于晦涩（优美的代码应当是明了的，命名规范，风格相似）

简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）

复杂胜于凌乱（如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁）

扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）

间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）

可读性很重要（优美的代码是可读的）

即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）

不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except:pass 风格的代码）

当存在多种可能，不要尝试去猜测

而是尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）

虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido ）

做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）

如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（方案测评标准）

命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）

1.2. Python 3 vs Python 2

“Python 2.x is legacy, Python 3.x is the present and future of the language.”

https://wiki.python.org

Guido van Rossum, the original creator of the Python language (and the Benevolent Dictator For Life), decided to clean up Python 2.x properly, with less regard for backwards compatibility. Python 3.0 was released in 2008.

The improvements include better Unicode support and more consistent features. We’ll point out some of these improvements as we encounter them in this course.

However, companies have invested in a significant amount of Python software developed over the years. The disadvantage of breaking backwards compatibility is that this software no longer works with 3.x.

**Which Version do I use?**

**In this course, we'll use Python 3.4.1.**

If you are starting a new project and you can do what you want with Python 3.x, it’s the way to go. 3.x is under active development. All recent standard library improvements are only available by default in Python 3.x.

However there are a few minor downsides.

Most current Linux distributions and Mac OS X are still using 2.x as default. You'll need to install Python 3 and override these defaults.

Because it is newer, Python 3.x has less third party library support. So if you need to use a specific third party package that is not compatible with Python 3, and porting that package is a non-trivial task, you may choose to use Python 2.

**1.3. Setting Up Python**

You need to install Python on your personal computer. In this course, we'll use version 3.4.1. We'll also install PyCharm Community Edition, a free lightweight Python Integrated Developmemt Environment (IDE).

PC Installation Instructions

**Step 1: Determine which version of Windows your PC is running:**

If you have Windows 8, open File Explorer then select This PC and click on Properties as shown below to determine the system type.

**Step 2: Install Python 3.4.1:**

Open the following link in a new tab: http://www.python.org/download/

If you have a 32-bit version of Windows, then select the Python 3.4.1 Windows x86 MSI Installer. Download the installer then double click on it to run it and follow the directions.

If you have a 64-bit version of Windows, then select the Python 3.4.1 Windows X86-64 MSI Installer. Download the installer then double click on it to run it and follow the directions.

**Step 3: Install PyCharm Community Edition:**

Open the following link in a new tab: http://www.jetbrains.com/pycharm/download/

Click on Download Community to download then run the PuCharm installer.

MAC Installation Instructions:

Your Mac may already have an earlier version of Python installed. You will need to install version 3.4.1.

Step 1: Determine your OS X version:

Determine your MAC OS X version by clicking on the link below and following directions.

http://support.apple.com/kb/ht1633

Write down your findings: I am running MAC OS version \_\_\_\_\_\_\_.

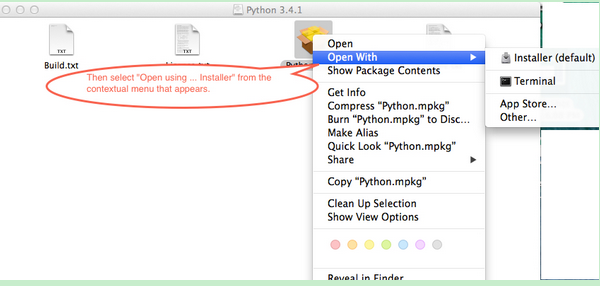
Step2: Install Python 3.4.1:

Open the following link in a new tab: http://www.python.org/download/

If you have Mac OS X version 10.6 and later, then select the Python 3.4.1 Mac OS X 64-bit/32-bit x86-64/i386 Installer. Download the installer then run it and follow the directions.In the readme.txt file, you'll see the following:

If you are attempting to install on an OS X 10.8+ system, you may see a message that Python can't be installed because it is from an unidentified developer. This is because this Python installer package is not yet compatible with the Gatekeeper security feature introduced in OS X 10.8. To allow Python to be installed, you can override the Gatekeeper policy for this install. In the Finder, instead of double-clicking, control-click or right click the "Python" installer package icon. Then select "Open using ... Installer" from the contextual menu that appears.





Otherwise if you have MAC OS version 10.5 and later, then select the Python 3.4.1 Mac OS X 32-bit i386/PPC Installer . Download the installer then run it and follow the directions.

Step 3: Install PyCharm Community Edition:

Open the following link in a new tab: http://www.jetbrains.com/pycharm/download/

**1.4. The Python Interpreter**

Now we can start practicing with the Python interpreter. We can do that in several ways:

1. By starting PyCharm and running the Python Console. The following screencast illustrates how to do that.
2. By invoking the interpreter from a terminal window on a Mac, or a command prompt window on a PC. To do that, we open the window and type python at the prompt. On a Mac, if you also have an older version of python installed, type python3 instead. The following screencast illustrates how to do that

From here on, instead of passively reading along, you should start the interpreter and try to replicate the actions.

You can type Python code directly at the '>>>' prompt. Make sure that you are accessing python 3.4.1. The version number appears right above the prompt.

Whenever you enter a complete code fragment, it will be executed. For instance, typing:

5 + 23

and pressing Enter will cause the following to be displayed:

28

>>>

Now type

print('Hello World!')

and hit Enter, you get:

Hello World!

To get some help, we type:

help()

We get the following:

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules.  To quit this help utility and >return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type...

...

help>

At the help> prompt, we can type print to get help on the print function or quit to go back to the

Python interpreter.

help> quit

Back at the >>> prompt, we can also type:

help(print)

to get help on the print function.

>>> help(print)

Help on built-in function print in module builtins:

print(...)

    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.

    Optional keyword arguments:

    file:  a file-like object (stream); defaults to the current sys.stdout.

    sep:   string inserted between values, default a space.

    end:   string appended after the last value, default a newline.

    flush: whether to forcibly flush the stream.

>>>

**1.5. IDLE**

IDLE is the standard Python development environment. It is installed when you install Python.

Its name is an acronym of "Integrated DeveLopment Environment" **IDE** but it’s actually named after the Monty Python comedian Eric Idle.

IDLE has a Python shell window, which also gives us access to the Python interactive mode.

**When you start IDLE, make sure you are accessing Python 3.4.1.** The version number appears right above the Python prompt in the shell window.

IDLE also includes a file editor to create, edit and run Python source files. In this course, I recommend using the PyCharm Community Edition IDE instead to take advantage of some useful features in code completion and debugging.

2. Python Basics

2.1. Numbers & Arithmetic Operators

As far as numbers are concerned, Python supports:

* Integers such as 9, -2 and 0
* Floats such as 5.0 and -4.5
* Complex numbers such as complex(1, 2) which denotes 1 + 2j

Let's go back to the interpreter and try a few things:

>>> 67 \* 23

1541

The basic mathematical operators +, -, \* (multiplication), / (division), // (floor division), % (modulo or remainder) and \*\* (exponent)指数 are built into the Python language. This means we can use them right away. If you want to use some more advanced mathematical functions such as trigonometric functions in your calculation, you need to import the math module. Do not worry about what it means right now. We will cover this later during the course.

Let’s explore the **order of operations**:Order of operations: \*\* \*, /,// +, - **指数→乘/除/取商数,→加/减**

>>> 1 + 2 \* 3

7

>>>3 \* 2 - 5

1

>>>1 + 2 \*\* 2

5

\*\* is the exponent (power) operator: 2 \*\* 2 is 2 to the power 2

>>> -1 + 2 \* 2

3

>>> 12 % 5

2

% is the modulo operator also known as the remainder: 12 % 5 is 2 which is the remainder you get when you divide 12 by 5.

Operators are evaluated using the standard order of operations. You can use parentheses to force certain operators to be evaluated first.

>>> (1 + 2) \* 3

9

**>>>-2 \* 3 \*\* 2 3 is first raised to the power 2 and then the result 9 is multiplied by -2**

-18

**Division and Integer Division:**

>>>10/2

5.0

In Python3 the result of **dividing** 2 integers is **NOT an integer**. So 10/2 returns 5.0 and 9/2 returns 4.5. In Python2, the result of dividing two integers was an integer; in other words 9/2 would have returned 4.

>>>9/2

4.5

>>>7//2 **//是用来取商数, 抛去小数位**

3

To perform an integer division and get an integer result, discarding any fractional result, there is another operator, //. Note that integer division returns the floor integer of the result.

>>>7//-2

-4

**round()** is a very useful built-in function when it comes to manipulating numbers. It allows to round a given number to a specified number of digits. Let's see how it works:

>>> round(5.156, 2)

5.16

>>> round(5.156, 1)

5.2

>>> round(5.156)

5

**type()** is a very handy Python function that returns the object type.相当于javascript中typeof

>>> type(9/2)

<class 'float'>

>>> type(3.337)

<class 'float'>

>>> type(5+3)

<class 'int'>

>>> type((5+3.0)) 整数和小数运算=小数

<class 'float'>

>>> type(-7//2)

<class 'int'>

>>> type(7\*\*2)

<class 'int'>

**2.2. Booleans, Logical & Comparison Operators**

Booleans are either True or False.

There are three logical operators defined on Booleans: and, or, not.

>>> not Ture

False

>>> True and False

False

>>> True or False

True

>>>False or False

False

**Familiar rules of Boolean algebra apply**

not True False 非真即假

not False True 非假即真

True and True True 真真为真

True and False False 真假货是假

False and True False 假真货是假

False and False False 错上加错是错

True or True True 真或真是真

True or False True 真或假是真

False or True True 真或假是真

False or False False 假货当然假

**The logical operators are evaluated in the following order:**

1. **not**
2. **and**
3. **or**

>>>True and not False = True and True

True

True and not False is the same as True and (not False)

>>>True or False and True = Ture or false

True

True or False and True is actually True or (False and true=false).

**Python has a boolean interpretation for non boolean values.**

For example **any nonzero number** is interpreted as **True**.

0 is interpreted as False.

>>> not 0 = not false

True

>>> not 6

False

**Python supports the following comparison operators:**

= = equal

**!= not equal**

> strictly greater than

< strictly less than

>= greater than or equal

<= less than or equal

Is 5 > 3?

>>>5 > 3

True

Is 2 <= 1?

>>> 2 < =1

False

>>> 200.2 <= 989.1

True

>>> 0 >= -16

True

**2.3. Variables & Assignments**

Variables are used to refer to information that can change over time. A variable has a name and that name is used to access the information.

>>> result = True

>>> result

True

**The = sign is an assignment operator which says: create a variable, call it result and assign this value (True) to it. You don't have to declare the variable first.**

The left side of an assignment statement has to be a variable name.

True = result is not the same as result = True.

**True = result is not allowed in Python.**

>>> True = result

File "<interactive input>", line 1

SyntaxError: assignment to keyword

>>> 5 = a

File "<interactive input>", line 1

SyntaxError: can't assign to literal

>>> result = True

You may be used to thinking about variables as containers for storing information. In Python, it is more correct to think of a variable as referring to the assigned value. This will make more sense with fancier data types. result refers to True here.

>>> not result

False

>>> type(result)

<class 'bool'>

**In Python, you don’t have to declare a variable type before using it.** The variable takes the type of whatever value you assign to it. Python figured out that result is Boolean because we assigned the value True to it.

>>> result = 5

>>> type(result)

<class 'int'>

Now result is assigned the integer value 5. No error is generated. result is now of type integer. result now 'refers' to 5 instead of True.

**Variable names** can contain both letters and numbers, **but they have to begin with a letter**. Other than that and the fact that **you cannot use a keyword as a variable name**, names don’t matter for correctness. They do matter for readability.

There are guidelines in the 'Style Guide for Python Code':

**Variable names should be lowercase**, with words separated by underscores as necessary to improve readability.

Example: homework\_grade, final\_grade

If you specify an illegal variable name, you’ll get an error:

>>> 5result = 3

SyntaxError: invalid syntax

>>> if = 4

SyntaxError: invalid syntax

**2.4. Constants?**

A constant, unlike a variable, refers to an identifier whose value is not supposed to change.

Python does not provide any mechanism to recognize constants or treat them differently than variables.

The convention is to **use uppercase letters for the constant names** so they are easily recognized as constants. However Python will not generate any error if we change their value.

When a constant name includes more than one word, we separate them with underscores.

CAPACITY = 50

MAX\_HEIGHT = 10

It is good practice to use explicit constants in our programs to make our code more readable and maintainable.

**2.5. Comparison or Assignment?**

**It is important to distinguish between the assignment operator (=) and the comparison operator (==).**

>>> counter = 2

counter is a variable. It is assigned the integer value 2. = is an assignment operator.

>>> counter = = 0

False

= = is a comparison operator. Is counter equal to 0?

>>> counter

2

counter is still 2. The expression above (= = ) has not changed its value.

>>> counter = 5

counter is now assigned the value 5.

>>> counter != 0

True

!= (not equal) is also a comparison operator

**2.6. Strings - The Basics**

Strings are sequences of Unicode characters.

Strings may be enclosed in:

* single quotes: 'Hi'
* double quotes: "Hello"
* triple quotes: '''Howdy'''

**Single quotes allow embedded double quotes:** ' "I love Python", she said.'

**Double quotes allow embedded single quotes**: "Isn't great?"

**Triple quotes allow strings to span multiple lines.**

Let's go back to the interpreter and type the following:

In Python3, print is a function. In Python2, print was a keyword. This would have been: print "Hello World"

>>> print("Hello World")

Hello World

friend is a variable. It refers to the string 'Bob' .

>>> friend='Bob'

>>> print("Hello", friend)

Hello Bob

--- Here print takes 2 strings, and prints them with a space in between. Friend is replaced by its value, 'Bob'.

>>> len(friend)

3

**--- len() returns the length of an object**, in this case the string 'Bob'. **Make sure you don’t use len as a variable name.**

>>> type(friend)

<class 'str'>

**'' (or "") denotes the empty string:**

>>> len('')

0

Let’s introduce the **string operation concatenation +.**

**+ just concatenates the two strings 'Foot' and 'hill'** with no space in between.

>>> college= 'Foot'+'hill'

>>> print(college)

Foothill

Make sure you **don’t mix strings and numbers** when you use +. You’ll get an error. **In python, you can not concatenate different datatypes**

>>> print("Python"+3)

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly

**String Indexing:**

Characters in a string can be accessed using the standard [ ] syntax. **Python uses zero-based indexing.**

>>> friend = 'Bob'

>>> friend[0]

'B'

>>> friend[1]

'o'

>>> friend[2]

'b'

If the index is out of bounds for the string, Python raises an error.

>>> friend[3]

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

IndexError: string index out of range

**Strings are immutable!**

Even though we can access individual characters in strings, we cannot modify them.

>>> friend = 'Bob'

>>> friend[0] = 'A'

Traceback (most recent call last):

File "<string>", line 301, in runcode

TypeError: 'str' object does not support item assignment

**What we can do is reassign the variable friend to a different string**:

>>> save\_friend = friend

>>> friend = 'Alice'

>>> save friend

'Bob'

Here we have not changed the string Bob to Alice. We have just made the variable friend point to the string Alice instead of Bob. save\_friend is still pointing to Bob.

**String Slices**

The "slice" lets us refer to parts of the strings. Later we’ll also see it with lists.

The **slice** **s[start:end]** consists of the elements beginning at start and extending up to **but not including end**.

Foothill

0 1 2 3 4 5 6 7

-8-7-6-5-4-3-2-1

college='Foothill'

college[4:6] is 'hi' -- chars starting at index 4 and **extending up to but not including index 6**

college[4:] is 'hill' -- **omitting either index defaults to the start or end of the string**

college[:4] is 'Foot' -- omitting either index defaults to the start or end of the string

college[:] is 'Foothill' -- omitting both always gives us a copy of the whole thing

college[2:100] is 'othill' -- an index that is too big is truncated down to the string length

**Negative index numbers count back from the end of the string:**

college[-1] is 'l' -- last char (1st from the end)

college[4:-2] is 'hi' – starting at index 4 going up to but not including the last 2 characters.

college[-4:] is 'hill' – starting with the 4th char from the end and extending to the end of the string.

college[:-1] is 'Foothil' – starting from the beginning of the string and omitting the last character.

**2.7. Conversion Functions**

So far we’ve covered integers, floats, booleans and strings.

The built-in function type() can help us confirm the type of an object.

>>> type (-300)

<class 'int'>

>>> type (5.76)

<class 'float'>

>>> type (5 == 0)

<class ‘bool'>

>>> type ('hello')

<class 'str'>

There are also a number of **built-in functions that help us convert** from one type to another: **int(),** **float(),** **str()** and **bool().**

**int()** is especially useful to convert user input from string to integer.

You’ll need to use it in the first programming assignment.

>>> a = '3'

>>> type (a)

<class 'str'>

>>> b = int (a)

>>> b

3

>>> type (b)

<class 'int'>

int () also allows us to convert a float to an integer.

>>> int (5.8)

5

>>> int (-5.8)

-5

Note that int () truncates towards 0.

Interestingly, if you convert True to integer, you get 1 whereas False gives you 0.

>>> int (True)

1

>>> **int (False)**

**0**

Similarly float() allows us to convert integers, strings or Booleans to floats.

>>> float (-3)

-3.0

>>> float ('-4.6')

-4.6

>>> float (True)

1.0

>>> float (False)

0.0

If you try to convert a non-numeric string, you’ll get an error.

>>> float ('hello')

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

ValueError: could not convert string to float: 'hello'

str() lets you convert anything else to a string.

>>> str (True)

'True'

>>> str (98)

'98'

>>> str (-6.3+2) # **先计算 然后转换**

'-4.3'

bool() lets you convert any value to its boolean interpretation:

0 is interpreted as False, any other number is interpreted as True.

>>> bool (98)

True

>>> bool (0)

False

Similarly the empty string is interpreted as False, any other string is interpreted as True.

>>> bool ('hello')

True

>>> bool ('')

False

**2.8. Input and Output**

In our programs, we'll often need to get some input from the user. The built-in function input allows us to prompt the user for input. We specify the prompt between the parentheses.

In Python 2, this function was called raw\_input.

>>> my\_name = input('Please enter your name: ')

Please enter your name: Rula

The function then reads whatever you enter into a string and that string is assigned to the variable my\_name. Note that we could have used any other variable than my\_name.

The variable my\_name now can be used to access the user input.

For example we can use it to print a hello message, using the print function.

>>> print("Hello", my\_name)

Hello Rula

It's important to **note that the input function returns a string** even if the user enters a number or a boolean. If we are expecting a number, we need to use a conversion function (int or float) to convert it.

In the example below we prompt the user for their age. We assume the user will enter a number. We'll show how to deal with error cases later in this course.

age\_string = input('Please enter your age: ')

We convert the string entered (age\_string) to an integer and save it to the variable age.

**age = int(age\_string)**

We increment the age variable by adding 1 to it.

age = age + 1

We can think of the assignment above as: the new value of the variable age gets assigned the old value of the variable age + 1. Another way to write that in Python is: age += 1

We are now ready to print out the user's age in a year:

print('Your age in a year will be: ', age)

Now let's explore the built-in function print. We have already seen that we can use is to output one or more values to the console or terminal. These values are usually separated by a space character.

>>> print(1, 2, 3)

1 2 3

We can give the print function values (such as 1, 2, 3, 'Hello') or variables such as my\_name, age, or result. The variables get replaced by their values and these values get printed.

>>> result = True

>>> print(result)

True

Note that if we want the values printed to be separated by some other character than space, we can specify the separator as follows:

>>> print (1, 2, 3, **sep = '+')**

1+2+3

If we don't want anything separating the values, we set the separator to the empty string:

>>> print (1, 2, 3, sep = '')

123

**2.9. Our First Python Program**

Now we're ready to create our first Python program.

In PyCharm, create a new project then click on File -> New, then select File. We'll name our program firsthello.py. Python programs have to be created with the file extension py.

Our first program will consist of the traditional print(‘Hello World’) and some comments.

**2.10. Hello Friend**

Now let’s modify our firsthello program so that it prompts the user for their name and prints a personalized greeting.

#-------------------------------------------------------------------------------

# Name: firsthello

# Purpose: our first Python program

#

# Author: Rula Khayrallah #

# Created: 09/14/2014

# Copyright: (c) Rula Khayrallah 2014

#-------------------------------------------------------------------------------

"""

A little Python program that prints a greeting

Prompt the user for their name.

Print a customized Hello message.

"""

name = input( name = input('Please enter your name: ') # prompt the user for their name print ('Hello', name) # print a personalized greeting

**2.11. Comments**

There will always be a time in which you have to return to your code, to fix a bug, or to add a new feature. Regardless, looking at your own code after six months is almost as bad as looking at someone else's code. What you need is a means to leave reminders to yourself as to what you were doing. For this purpose, you leave comments. Comments are little snippets of text embedded inside your code that are ignored by the Python interpreter.

**A comment starts with the hash character (#) and extends to the end of the line.**

In our firsthello.py program, we included block comments at the top of the file to document its use:

#-------------------------------------------------------------------------------

# Name:        firsthello

# Purpose:     our first Python program

#

# Author:      Rula Khayrallah

#

# Created:     09/14/2014

# Copyright:   (c) Rula Khayrallah 2014

#--------------------------------------------------------------------------------

We also included inline comments in the code, although in this simple program they were not necessary.

name = input('Please enter your name: '**) # prompt the user for their name**

print ('Hello', name**) # print a personalized greeting**

**These comments are for your own use or for a programmer looking at your source code. You use them to include implementation details that are not obvious.**

**2.12. Documentation Strings**

But what if you just want to know how to use a program (or function, class or method) without looking at the code? **Documentation strings (or docstrings) are used to create easily-accessible documentation.** You can add a docstring to a module, function, class by adding a string **as the first statement.**

The convention is to use **triple-quoted strings**, because it makes it easier to add more documentation spanning multiple lines.

"""

A little Python program that prints a greeting

Prompt the user for their name.

Print a customized Hello message.

"""

To access this documentation, you can now use the help function inside a Python shell with the object you want help on.  **Just make sure the module has been imported first.**

For example, in the PyCharm console, type:

import firsthello

Then type:

help(firsthello)

You'll see the following:

**Note that the first line of the docstring appears as an overview of the module under NAME.**

**The first line is followed by an empty line and all subsequent lines appear under DESCRIPTION.**

**Docstrings are aimed at the user of your module. You describe the expected input, output and overall behavior.**

**2.13. A Tip Calculator**

Now we're ready to write a more practical program. We'll make it a tip calculator. The program will prompt the user for the cost of their meal, will calculate the tip amount (assuming a 20% tip) and the total cost of the meal and will print them out. To keep the program simple, we'll ignore any taxes on the meal.

#-------------------------------------------------------------------------------

# Name:        tip

# Purpose:     tip calculator

#

# Author:      Rula Khayrallah

#

# Created:     09/14/2014

# Copyright:   (c) Rula Khayrallah 2014

#-------------------------------------------------------------------------------

"""

Tip calculator assuming a 20% tip rate

Prompt the user for the cost of their meal.

Print the tip amount and the total cost.

"""

TIP\_RATE = 20/100         # declare the tip rate constant: 20%

user\_input = input('Please enter the cost of your meal in $: ')

cost = float(user\_input)  # convert the input string to a number

tip = TIP\_RATE \* cost     # calculate the tip amount

tip = round(tip, 2)       # round tip to two decimals

print('Tip Amount: $', tip, sep = '')   # suppress the space separator

total = cost + tip        # the total cost of the meal

total = round(total, 2)   # round total to two decimals

print('Total cost: $', total, sep = '') # suppress the space separator

Once we enter our program and save it (tip.py), we can run it as follows:

 Please enter the cost of your meal in $: 27.89

Tip Amount: $5.58

Total cost: $33.47

**3.1. What is a List?**

**Python has a very useful built-in type named list**. It is similar to **arrays 数列矩阵** in other programming languages.

A list is written as **a series of comma-separated values (items) between square brackets**. List items need not all have the same type.

>>> my\_list = ['Shakespeare', 500, 3.2, True]

>>> my\_list

['Shakespeare', 500, 3.2, True]

**We can get the length of a list by using the same built-in function len() that we used for strings.**

>>> **my\_list=['Shakespeare', 500, 3.2, True]**

**>>> len(my\_list) 显示一个list里有几项**

**4**

**The "empty list" is just an empty pair of brackets [].**

**>>> nothing = [ ]**

**>>> nothing**

[]

>>> len(nothing)

0

**3.2. Indexing and Slicing**

List indexing is similar to string indexing. We use the square brackets [ ] to access the items in the list, with the **first item at index 0 and the last item at index -1.**

>>> my\_list = ['Shakespeare' , 500 , 3.2, True]

>>> my\_list [0]

'Shakespeare'

>>> my\_list[3]

True

**Negative index numbers count back from the end of the string:**

>>> my\_list[-1]

True

>>> my\_list[-2]

3.2

We get an exception (error) when we specify an index out of range.

>>> my\_list[5]

Traceback (most recent call last):

File "<input>", line 1, in <module>

IndexError: list index out of range

**We can get a slice of the list**. **my\_list[start: stop]** is a sub-list that is comprised of the items **at start up to** **but not including the item at stop**.

>>> my\_list[1:3]

[500, 3.2]

**Omitting either index defaults to the start or end of the list.**

>>> my\_list [1:]

[500, 3.2, True]

>>> my\_list [:-1]

['Shakespeare', 500, 3.2]

**Omitting both indexes gives us all the items in the list.**

my\_list [:]

['Shakespeare', 500, 3.2, True]

**3.3. Concatenating Lists**

**We can use the + operator to concatenate lists as we did for strings**.

>>> my\_list = ['Shakespeare', 500, 3.2, True]

>>> **new\_list = my\_list + ['red','white','blue']**

['Shakespeare', 500, 3.2, True, 'red', 'white', 'blue']

>>> my\_list

['Shakespeare', 500, 3.2, True]

**The original list, my\_list is unchanged**.

**We can also use the \* operator to repeat lists.**

>>> music\_notes = ['C', 'C', 'G', 'G', 'A', 'A', 'G']

>>> song = 4 \* music\_notes

>>> song

['C', 'C', 'G', 'G', 'A', 'A', 'G', 'C', 'C', 'G', 'G', 'A', 'A', 'G', 'C', 'C', 'G', 'G', 'A', 'A', 'G', 'C', 'C', 'G', 'G', 'A', 'A', 'G']

**4\* This is equivalent to concatenating the list music\_notes four times**: (music\_notes + music\_notes + music\_notes + music\_notes).

>>> music\_notes

['C', 'C', 'G', 'G', 'A', 'A', 'G']

The original list music\_notes is still unchanged.

**3.4. Modifying Lists**

Unlike strings, which are immutable, **it is possible to change individual elements of a list**.

>>> my\_list=['Shakespeare', 500, 3.2, True]

>>> my\_list[2]=0

>>> my\_list

['Shakespeare', 500, 0, True]

**Assignment to slices is also possible:**

>>> my\_list

['Shakespeare', 500, 0, True]

>>> my\_list[0:2]=['apples','oranges']

>>> my\_list

['apples', 'oranges', 0, True]

We modified my\_list[0] and my\_list[1] in one step.

**Assignment to slices can even change the size of the list** :

>>> my\_list

['Shakespeare', 500, 0, True]

**>>> my\_list[0:2]=[] //remove items at 0, 1 position**

**>>> my\_list**

**[0, True]**

We effectively removed the items at index positions 0 and 1 from the original list.

We can also clear the list as follows:

**>>> my\_list[:] = [] remove everthing**

>>> my\_list

[]

**3.5. Nested Lists**

We can also create lists that contain other lists:

>>> exams = [85, 90]

>>> grades = [100, 98, exams, 85]

>>> grades

[100, 98, [85, 90], 85]

>>> len(grades)

4

The list grades has 4 items, not 5. The nested list counts as one item regardless of how many items it contains.

The item at index position 2 in grades is a list.

>>> grades[2]

[85, 90]

To access an item inside the nested list we can do the following:

>>> item = grades[2]

>>> item[0]

85

We can also combine the two statements above into one as follows:

>>> grades[2][0]

85

**3.6. Membership Test**

We can test if an item is in the list using in and not in:

>>> grades = [100, 98, [85, 90], 85]

>>> 60 in grades

False

>>> 100 in grades

True

>>> 75 not in grades

True

>>> 90 in grades

False

90 is a member of the **nested list**, not of grades.

>>> 90 in grades[2]

True

**3.7. Copying Lists**

Remember our discussion about variables ‘referring’ to objects as opposed to ‘containing’ them?

Assignment with an = on lists does not make a copy.

Instead, the assignment makes the two variables point to the same list in memory.

>>> alice=[98,87,100]

>>> **bob=alice**

>>> alice

[98, 87, 100]

The two variables **alice** and **bob** point to the same location in memory.

>>> bob

[98, 87, 100]

What happens if we now change one list item in alice?

>>> alice[1]=0

>>> alice

[98, 0, 100]

>>> bob

[98, 0, 100]

**What if we wanted bob to have a different copy of list, one that initially has the same values as alice but that is not affected by future changes to alice.**

**>>>> alice = [98, 0, 100]**

**>>> bob = alice[:]**

**----The two variables alice and bob point to different locations in memory**

**>>> bob**

**[98, 0, 100]**

**>>> alice**

**[98, 0, 100]**

>>> alice[1]=5

>>> alice

[98, 5, 100]

**>>> bob**

**[98, 0, 100]---- bob is unchanged**

However with nested lists the slice assignment will NOT work.  It is a **shallow copy.**

>>> charlie = [100, 98, [85, 90], 85]

>>> diana = charlie[:]

>>> diana

[100, 98, [85, 90], 85]

>>> charlie[1] = 0

>>> charlie

[100, 0, [85, 90], 85]

>>> diana

[100, 98, [85, 90], 85]---原本中**非nested**的数变了, shallow copy的不做任何变 因为里面有nested list

>>> charlie[2][1] = 0

>>> charlie

[100, 0, [85, 0], 85]

>>> diana

[100, 98, [85, 0], 85]---但是可以改变 nested list 中的数.

We'll later see how to use deepcopy to get around this problem.

**3.8. What else can we do with a list?**

**If the list items are comparable**, we can get their minimum, their maximum and get the list in sorted order.

If the items are numbers, we can also get their sum.

>>> grades=[85, 60, 100, 95, 75]

>>> min(grades)

60

>>> max(grades)

100

>>> sum(grades)

415

>>> sorted(grades)

[60, 75, 85, 95, 100]

>>> grades

[85, 60, 100, 95, 75]

Note that the function **sorted() does not change the original list**. It returns a sorted copy of the original list.

min(), max(), sum and sorted() are all built-in functions. We invoke them on a list by specifying the list between the parentheses: max(grades), min(grades), sorted(grades).

There are also many list methods. We invoke these methods by adding a dot following the method name right after the list as shown below. We'll cover methods in details later in this course,

>>> grades = [85, 60, 100, 95, 75]

**append** is a list method that **adds a new element** to the **end** of a list.

>>> grades.append(100)

>>> grades

[85, 60, 100, 95, 75, 100]

**reverse** rearranges the elements **in reverse order**.

>>> grades.reverse()

>>> grades

[100, 75, 95, 100, 60, 85]

**sort** rearranges the elements in **ascending order.**

>>> grades.sort()

>>> grades

[60, 75, 85, 95, 100, 100]

insert adds an element at a given index position, and push the original data to the right To insert 90 at index position 0, we write:

>>> grades.insert(0, 90)

>>> grades

[90, 60, 75, 85, 95, 100, 100]

**Note that all these methods (append, reverse, sort and insert) change the original list.**

**index** returns **the position of the first occurrence** of the given element.

>>> grades.index(75)

2

**count** returns the **number of occurrences** of the given element.

>>> grades.count(100)

2

We can also remove items from a list:

If we know the index of the element we want, we can use **pop**: **it modifies the list and returns the element that was removed.**

>>> grades.pop(2)

75

>>> grades

[90, 60, 85, 95, 100, 100]

75 is no longer an item in grades.

**If we don’t provide an index, pop deletes and returns the last element.**

>>> grades.pop()

100

>>> grades

[90, 60, 85, 95, 100]

**If we don’t need the removed value, we can use the del operator:**

>>> del grades[1]

>>> grades

[90, 85, 95, 100]

**If we know the element we want to remove but not the index, we can use remove:**

>>> grades.remove(85)

>>> grades

[90, 95, 100]

Note that **only the first instance of the given element is removed**.

>>> grades = [90, 60, 85, 95, 100, 100]

>>> grades.remove(100)

>>> grades

[90, 60, 85, 95, 100]

One way to see all the methods available on lists is to use code completion in the interpreter shell or the editor. Typing grades followed by a dot brings up a list of all the methods available on grades:

**3.9. From strings to lists ... and back**

Sometimes it is useful to be able to **separate some text into a list of words**.

**split()** allows to do just that. It is a string method so we invoke it on a string and it returns a list of the words in the string.

Let's see how it works with an example:

phrase = 'Simple is better than complex'

words = phrase.split()

print(words)

['Simple', 'is', 'better', 'than', 'complex']

Having the words in a list allows us to traverse them with a for loop and operate on each word individually.

for word in words:

print(word )

Simple

is

better

than

complex

The **join()** method allows us to go back from a list to a single string. It joins the elements of a list into a string with each element separated by the string that join was invoked on.

new\_phrase = '**\*'.join(words)**

print(new\_phrase)

Simple\*is\*better\*than\*complex

Or we can invoke the join on a single space character:

new\_phrase = ' '.join(words)

Simple is better than complex

We should note here that **split also allows use to specify a delimiter other than white space**:

Here we specify '\*' as the delimiter for the split/

volume = 'height\*weight\*depth'

dimensions = volume.split('\*')

print(dimensions)

['height', 'weight', 'depth']

|  |
| --- |
| **4.1. Conditional Statements** |
|  |
|  |
| Let's go back to our first hello program and modify it as follows:  if the user does not enter a name, we would like our program to print:  Hello Friend.  To do that, we need an if statement:  #-------------------------------------------------------------------------------  # Name:        firsthello  # Purpose:     our first Python program  #  # Author:      Rula Khayrallah  #  # Created:     09/14/2014  # Copyright:   (c) Rula Khayrallah 2014  #-------------------------------------------------------------------------------  """  A little Python program that prints a greeting    Prompt the user for their name.  Print a customized Hello message.  """  name = input('Please enter your name: ')  # prompt the user for their name    **if**name:                  # check that a name was entered - **same as name !=''**      print('Hello', name)    # print a personalized greeting  **else:**      print('Hello Friend')   # print a generic friendly greeting    Let’s look carefully at our modified program.  We have an **if-else** **conditional statement.**  The syntax is as follows:  **if** *condition***:**      indentedStatementBlockForTrueCondition  **else:**      IndentedStatementBlockForFalseCondition  These statement blocks can have any number of statements, and can include about any kind of statement.  The condition may be any expression that evaluates to True or False.  The following are all examples of valid conditions:  if name not in class\_list:  if  100 in grades:  if result < 40 or result > 100:  We can also have a simple if statement with no else part.  if name == 'Bob':      print('How are you?')      print('Long time no see')  The general Python syntax for a simple if statement is  if *condition* :      indentedStatementBlock  If the condition is true, then do the indented statements.  If the condition is not true, then skip the indented statements.  As with other kinds of statements with a heading and an indented block, the block can have more than one statement.  In the example above the choice is between doing something (if it’s Bob) or nothing (if it’s not Bob).  Often there is a choice of more than 2 possibilities.  Here’s where the if-elif syntax comes in handy.  **elif** stands for else if.  But **you have  to use elif.  else if will NOT work.**  The general syntax is as follows:  if condition1 :      indented Statement Block For **First** True Condition1  elif condition2 :      Indented Statement Block For **First** True Condition2  elif condition3 :      indented Statement Block For **First** True Condition3  elif condition4 :      indented Statement Block For **First** True Condition4  else:     indented Statement Block For Each Condition False  With this construction **exactly one of the indented blocks is executed**. It is the one corresponding to the first True condition.  If all conditions are False, the block after the final else line is executed.  Let's write a program that will look up the tax rate based on the taxable income.  The tax brackets are defined as follows:       |  |  | | --- | --- | | Taxable Income | Tax Rate | | $89,351 and above | 0.28 | | $36,901 to $89,350 | 0.25 | | $9,076 to $36,900 | 0.15 | | below $9,076 | 0.10 |     #-------------------------------------------------------------------------------  # Name:        taxrate  # Purpose:     demonstrate the use of elif  #  # Author:      Rula Khayrallah  #  # Created:     09/15/2014  # Copyright:   (c) Rula Khayrallah 2014  #-------------------------------------------------------------------------------  """  Tax rate lookup    Prompt the user for the taxable income.  Print the corresponding tax rate.  """    user\_input = input('Please enter the taxable income: ')  income = float(user\_input)  # convert the input string to a number    **if income >= 89351:**  **tax\_rate = 0.28**  **elif income >= 36901:  # no need to check that income < 89351**  **tax\_rate = 0.25**  **elif income >= 9076:   # no need to check that income < 36901**  **tax\_rate = 0.15**  **else:**  **tax\_rate = 0.10**    print('The tax rate is:', tax\_rate)    With this construction **exactly one** of the indented blocks is executed.  Keep this elif syntax in mind when you are working on your programming assignment.  4.2. For Loops 回路  A for loop is a very useful construct in Python.  It allows us to **go over the items of a sequence such as a list or a string one by one. We also say we are iterating 重复over the sequence**.  The general syntax of a for loop is as follows:  **for variable\_name 可以起任何名字 in sequence\_name:**  **indented\_statement(s)**  **Other statements…(execute after loop finish)**    The **indented\_statements** (also **called the loop body**) are **executed several times, once for each item in the sequence.**    We'll see some examples with **lists, strings and range** next. |

**4.3. Going over a list 重复**

Let’s start with a list, and see how we can go over its items, one by one.

cs21 = ['Alice', 'Bob', 'Carol', 'Evan']

Our list, cs21, has 4 items. Each item is a string denoting a student name.

cs21 = ['Alice', 'Bob', 'Carol', 'Evan']

for student in cs21**:**

print(‘ Hello’, students)

print (‘ Welcome to CS21A’)

**student is just a variable name**. It refers to all the items in the list, one after the other.

**I could have used any other variable name.** I used student to remind myself that I am dealing with student names.

**Note the colon : after the list name.**

**Indentation** is what determines what gets done at each iteration of the loop. The **indented print statement** will be **executed with every iteration** whereas the *non indented print statement* will be *executed* ***after the loop is exited***.

When we run the program above, the following output is generated:

Hello Alice

Hello Bob

Hello Carol

Hello Evan

Welcome to CS21A

The table below traces the program step by step. The value of the variable student is shown in red at each iteration of the loop. When we are done with all the items in the list, we exit the loop. The next non indented statement is executed: print('Welcome to CS21A').

The table below traces the program step by step.  The value of the variable student is shown in **red** at each iteration of the loop. When we are done with all the items in the list,  we exit the loop.  The next non indented statement is executed: print('Welcome to CS21A').

|  |  |  |
| --- | --- | --- |
| cs21 | student | Print output |
| ['Alice','Bob','Carol','Evan'] |  |  |
| [**'Alice'**,'Bob','Carol','Evan'] | **'Alice'** | Hello **Alice** |
| ['Alice','**Bob'**,'Carol','Evan'] | '**Bob'** | Hello **Bob** |
| ['Alice','Bob',**'Carol'**,'Evan'] | **'Carol'** | Hello **Carol** |
| ['Alice','Bob','Carol',**'Evan'**] | **'Evan'** | Hello **Evan** |
|  |  | Welcome to CS21A |

Warning:

When you are iterating over a list, make sure you don't modify its elements. The outcome will be unpredictable if you do.

Here's an example of what **NOT to do**:

my\_list = [1, 2, 3, 4, 5]

for number in my\_list:

print(number)

my\_list.remove(number)

print(my\_list)

The corresponding output is:

1

3

5

[2, 4]

You can see that because we removed items from the list while iterating over it, some items (3. 7) were skipped.

**If you need to modify the list you are iterating over while inside the loop you should make a copy**. Iterate on the **copy** and **modify the original**. To make a copy, **the slice assignment may be used if the list is not nested**.

my\_list = [1, 2, 3, 4, 5]

**copy\_list = my\_list[:] //shallow copy**

for number in copy\_list: // going over the copy

print(number)

**my\_list.remove(number) // modify the original**

print(my\_list)

Now the output is:

1

2

3

4

5

[] ---loop完后的指令是print(my\_list),但是这个sequence里面的数已经被remove,所以返回的是空集

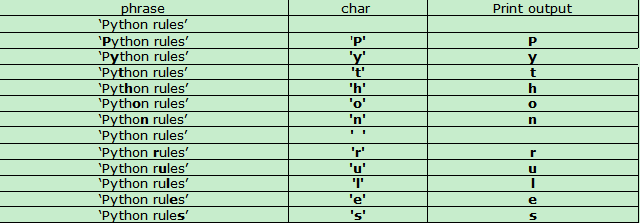
**4.4. Going over a string**

Going over a string

We can use the for loop in a similar manner with a string:

**Here again, char is an arbitrary variable name. I could have used letter, or x or my\_var**. I just used char to remind myself that I am dealing with a character.

The table below traces the program step by step. The value of the variable char is shown in red at each iteration of the loop.



Note that print adds a new line after each print invocation. So a new line is added after each character printed.

phrase='Python Programming'

for char in phrase:

print(char)

print(phrase)

The corresponding output:

P

y

t

h

o

n

P

r

o

g

r

a

m

m

i

n

g

Python Programming

|  |  |  |
| --- | --- | --- |
| phrase | char | Print output |
| ‘Python rules’ |  |  |
| ‘**P**ython rules’ | '**P'** | **P** |
| ‘P**y**thon rules’ | **'y'** | **y** |
| ‘Py**t**hon rules’ | **'t'** | **t** |
| ‘Pyt**h**on rules’ | **'h'** | **h** |
| ‘Pyth**o**n rules’ | **'o'** | **o** |
| ‘Pytho**n** rules’ | **'n'** | **n** |
| ‘Python rules’ | **'  '** |  |
| ‘Python **r**ules’ | **'r'** | **r** |
| ‘Python r**u**les’ | **'u'** | **u** |
| ‘Python ru**l**es’ | **'l'** | **l** |
| ‘Python rul**e**s’ | **'e'** | **e** |
| ‘Python rule**s**’ | **'s'** | **s** |

When we are done with all the characters in the string, we exit the loop. The next non indented statement is executed: print(phrase).

**4.5. Nested for Loops**

We can have one loop nested inside another loop.

In the example below, the outer loop goes over the elements of a list. These elements happen to be strings. So the inner loop goes over the characters in each of these strings.

**Indentation is what determines what statement goes with what loop.**

class\_list = ['Alice', 'Bob', 'Carol', 'Evan']

for student in class\_list:

    for char in student:

        print(char) // inner loop

    print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_') // orange color 是outer loop

print('Welcome to CS 21A')

 The outer loop will be executed 4 times.  Once for each student in the list.

class\_list = ['Alice', 'Bob', 'Carol', 'Evan']

for student in class\_list:

    for char in student:

        print(char) →红色是inner loop

    print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

print('Welcome to CS 21A')

What about the inner loop? How many times will it be executed?

Here's the corresponding output:

|  |  |
| --- | --- |
| |  | | --- | | A  l  i  c  e  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  B  o  b  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  C  a  r  o  l  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  E  v  a  n  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Welcome to CS 21A | |
|  |

**4.6. Range**

Syntax: **Range (startpoint, endpoint that is not included, increase/decrease unit each time)**

Now is a good time to introduce a useful construct in Python. The range is sequence type constructs a sequence of integers. We can use this sequence in the for loop, as we would a list.

Note that in Python 2, range was a built-in function that was used to create and return a list.

In Python 3, it returns a range object.

**range(3) -> 0, 1, 2 不给startpoint的情况下, 括号内数是ending point**

for i in range(3):

print(i) **## range of integers from 0 up to but NOT including 3**.

0

1

2

range(1,5) -> 1,2,3,4

for i in range(1, 5): **## range of integers from 1 up to but NOT including 5.**

print(i)

1

2

3

4

range(5, 52, 5) -> 5, 10, 15, 20, 25, 30, 35, 40, 45, 50

range of integers **from 5 up to but NOT including 52**, **incrementing by 5** each time.

The integers don’t have to be positive:

range (-10, -5) -> -10, -9, -8, -7, -6

for i in range(-10, -5):

print(i)

-10

-9

-8

-7

-6

range (5, -5,-2) -> 5, 3, 1, -1, -3

for i in range(5, -5, -2):

print(i)

5

3

1

-1

-3

range(2, -2, 1) -> nothing

for i in range(2, -2, 1):

print(i)

In general:

**range(stop)** -> Returns a sequence of integers from 0 up to but not including stop.

**range(start, stop)** -> Returns a sequence of integers from start up to but not including stop.

**range(start, stop, step)** -> Returns a sequence of integers from start up to but not including stop **incrementing/increasing** by step.

Note that start, **stop and step have to be integers**.

We can also test for membership in a range:

>>> pick = 9

>>> pick in range(10)

True

>>> pick in range(5)

False

>>> 5 in range(5)

False

**4.7. A Factorial Program Example**

**The factorial of a number (!) is the product of all the numbers from 1 up to and including the given number.**

**1! = 1**

**2! = 1 \* 2**

**3! = 1 \* 2 \* 3**

**And so on.**

**For I in range(1, n+1):**

**Factor= I \* factor**

**Let’s write a Python program that computes the factorial of a user entered number. Our program will illustrate the use of a for loop with the range function.**

**user\_input = input('Please enter a positive integer: ')**

**number = int(user\_input)**

**result = 1 # initialize the factorial result to 1**

**for i in range(1, number + 1): # omit 0 but include the number itself**

**result = result \* i**

**print(result)**

**Please enter a positive integer: 5**

**120**

**The table below traces the program step by step when the user enters 5.**

|  |  |  |
| --- | --- | --- |
| number | i | result |
| 5 |  | 1 |
| 5 | 1 | 1 \* 1 = 1 |
| 5 | 2 | 1 \* 2 = 2 |
| 5 | 3 | 2 \* 3 = 6 |
| 5 | 4 | 6 \* 4 = 24 |
| 5 | 5 | 24 \* 5 = **120** |

|  |
| --- |
| **4.8. While Loops** |
|  |
|  |
| A **while** loop **executes a statement block repeatedly** as long as a given **condition is true**.  **Syntax**  while *condition* :                  indentedBlock to execute if condition is true  statement to execute if condition is false  **Example 1:**  while True:      print(‘Ha’)  print(‘Bye’)  Do **NOT** try that.  This is an example of an infinite loop.  It does not terminate.  It never gets to print (‘Bye’).    **Example 2:**  while False:      print('this will never get printed')  print('Bye')  Bye  **The statement inside the loop will never get executed because it is false, 换句话说condition是ture的时候, 语句始终循环在回路内 一直跑圈. 如果condition is false,那就跳过ture statement, 直接exit loop and print the false/last statement**    **Example 3:**  counter = 1  while counter < 10:      print(counter\*\*2)  This is another infinite loop.  What is missing? counter is always 1.  We need to update counter inside the loop.  This is a very common mistake.  Remember to update the variable used in the condition.    **Example 4:**  counter = 1  while counter < 10:  **print(counter\*\*2)   # print the first 9 squares; counter\*\*2=counter²**  **counter += 1   # shorthand for counter = counter + 1**  1  4  9  16  25  36  49  64  81    **Example 5:**  counter = 1  while counter < 10:      print(counter)  **counter \*= 2   # shorthand for counter = counter \* 2**  1  2  4  8  **☆Example 6: ※**  Here's a more useful example of a while loop:  **more\_input = True ## set variable more\_input , give it Boolean value as True**  **while more\_input: ## while condition of more\_input**  **user\_input = input('Please enter a positive integer: ') ##prompt user\_input for a**  **positive#**  **number = int(user\_input) ##convert user\_input to a integer and set it as variable**  **if number > 0: ##测试if 条件 number>0**  **more\_input = False ##如果if条件成立, 把more\_input改为false, 即终**  **止more\_input loop.**  **print('Thank you') ## more\_input loop终止后打印出语句**  Here we keep prompting the user for input until they enter a positive integer. **If they do, we set the boolean variable more\_input to False**. **As a result, the while loop is exited and we print ‘Thank you’.**  **The loop is executed at least once because more\_input is initialized to True.** It may be executed more times, depending on the user input. The number of times it is executed is **NOT predetermined like in the FOR loop.**  Please enter a positive integer: -4  Please enter a positive integer: 0  Please enter a positive integer: 9  Thank you  With this set of user input (-4, 0, 9), the loop is executed 3 times.  ☆Example 7 ☆:  valid\_input = False ## set valid\_input to Boolean value False  while not valid\_input: ## 即 while valid\_input is True,用以 prompt 下面input  name = input('Please your name: ') ## when valid\_input is True prompt user for input  if name: # this is the same as name != "" **即name非空**  valid\_input = True # **If 条件成立,启用valid\_input=true, 这个必须与起始的**  **valide\_input 值相反,这样才能终止while loop打印出**  **最后一句**  print('Hello', name)  Here we keep prompting the user for input until they enter a non-empty string. If they do, we set the boolean variable valid\_input to True. As a result, the while loop is exited and we print the Hello message.  Here again, the loop is executed at least once because valid\_input is initialized to False.  If the user just hits Enter, we keep prompting them…  Please your name:  Please your name:  Please your name: Amy  Hello Amy |
|  |
|  |
| **4.9. Factorial Program - Revisited**  Let's implement our factorial program, using a while loop instead of a for loop. We'll also add some input validation to make sure the user enters a positive number. Note that we are assuming that the user enters an integer (not a float and not a string) at this time.  # Part 1: Prompt the user for a positive integer  valid\_input = False #变量初始值设置必须保证让第一个loop启用  while not valid\_input: # while 后面的原变量取值与初始值相反, 这样下面的首次while loop开启  user\_input = input('Please enter a positive integer: ') #第一个loop启动, prompt for input  number = int(user\_input) #convert to integer format  if number > 0: # if 条件 上面的number 是否>0  valid\_input = True # 如果if 条件成立, 则 反馈原变量**取与初始值相反用以终**  **止loop**  # Part 2: compute the factorial of the user entered integer  result = 1 # 设置结果变量=1  i = 1  while (i <= number):  result = result \* i  i += 1 # don't forget to update the loop variable, i = i +1  print(result) |
|  |

**ASSIGNMENT 2:**

**\_\_author\_\_ = 'edward'**

**valid\_input = True**

**while valid\_input:**

**user\_input = input('Please enter a grade:')**

**if user\_input == "":**

**import sys**

**sys.exit()**

**else:**

**grade = float(user\_input)**

**score\_list = [grade]**

**print(score\_list)**

**if grade < 0 or grade > 100:**

**valid\_input = False**

**print('invalid score')**

**5.1. What is a function?**

A **function** is a named sequence of statements.

When we **define** a function, we specify its name and the sequence of statements. Later, we can call the function by name and the sequence of statements gets executed.

In other words a function is a **block of code that is defined once and is executed every time we call it.**

We have already seen several examples of a **function call:**

print(), type(), len(), int(), str(), min(), max(), sorted(), sum()

We say that a function takes an argument and returns a result. The result is called **the return value**.

The function len takes a string as its argument and returns the number of characters in the string.

len('Hello') -> 5

DRY - Don't Repeat Yourself

The DRY principle is a central idea in software development.

You should not have multiple fragments of code that perform the same thing – even if you can copy and paste the code into different locations. That results in code that is harder to test and harder to maintain.

Instead, that logic should be implemented once, given a name, and called multiple times.

**5.2. Function Definition**

To create our own function, we write a function definition.

A function definition starts with the reserved word **def** (short for define), followed by the function name.

Here's a function definition for a function named area:

def area(length, width):

""""

Compute the area of a rectangle

**Parameters: 参量**

length, width (float)

Returns:

area (float)

"""

result = length \* width

return result

**Function names are usually lowercase**. The function name here is area.

Our function takes two input parameters: length and width. They are placed between parentheses after the function name.

The function definition header ends with a colon.

Everything in a function definition is indented. The first line that is not indented is outside the function. There are no curly braces and no begins and ends to delimit the function otherwise.

A function has its own docstring. It is the first thing in the function. It is also indented.

result is a variable that is l**ocal** to the function. // **only can be accessed within the function**

A function **may or may not** have a return statement. The **return** statement **causes the function to exit,** and **may or may not pass back an expression** (return value) **to the caller**.

Our function returns the value of the variable result which is the product of the length and width.

**If the return statement is omitted the function is exited at the end of the indented code and the** value **None is returned**. The value **None is also returned** when a return statement is included without an associated expression as in:

return

**5.3. Function Calls**

Once we define our function, we can call it with different values for the parameters length and width as follows:

bedroom = area(11, 9) # first function call

kitchen = area(12, 7) # second function call

family = area(12, 12) # third function call

print(bedroom, kitchen, family)

99 84 144

The **initial values given to the parameters** at the point of call are called the **arguments**.**给予参数**

In the first function call above, 11 and 9 are the **arguments** to our function. They define the value that will be assigned to the parameters, length and width inside the function definition.

In the second function call, 12 and 7 are the arguments and in the third function call 12 and 12 are the arguments.

The return value of a given function is accessed by writing **function\_name(arguments).**

To access the **return value** of the function area when it is called with arguments 11 and 9, I can write area(11, 9). The corresponding value is 99.

Note that a **return statement** is **not** a **print statement**. **Just because a value is returned does not mean it will be printed**. If we need the value to be printed, we'll add a print statement as in:

print(bedroom, kitchen, family)

We can also write:

print(**area(11, 9), area(12, 7), area(12, 12**))

We call this **function composition合成函数/复合函数 即把一个函数返回的值用作另外一个函数的参数**: we are directly using the return value from one function (area) as an input argument to another function (print).

We have already done this in statements such as:

print(len(name))

print(max(grades))

**5.4. Arguments and Parameters**

We've seen that **parameters are the variables in the function definition** and **arguments are the values given to the variables** **at the point of call**.

**Parameters appear in the function definition; arguments appear in the function calls.**

In Python, we can call a function using one of the following types of arguments:

* Required positional arguments
* Keyword arguments
* Default arguments (positional or keyword)

**Required positional arguments** are the arguments passed to a function in correct positional order. The number of arguments in the function call should match exactly the number of parameters in the function definition. **给予参数必须与parameter 数量,位置一致 否则报错**

Le't go back to our area function definition and call our function area with different arguments:

def area(length, width):

""""

Compute the area of a rectangle

Parameters:

length, width (float)

Returns:

area (float)

"""

result = length \* width

return result

print(area(8, 3))

24

print(area(4\*3, 1+2))

36

my\_length= 7

print(area(my\_length, 2))

14

print(area(2))

Traceback (most recent call last):

File "C:/Users...", line 30, in <module>

print(area(2))

TypeError: area() missing 1 required positional argument: 'width'

Here we provided one argument instead of two so we got a TypeError.

print(area(6, 2, 3))

Traceback (most recent call last):

File "C:/Users...", line 30, in <module>

print(area(6, 2, 3))

TypeError: area() takes 2 positional arguments but 3 were given

Here we provided three arguments instead of two so we also got a TypeError.

**Keyword Arguments:**

We can also call a function using keyword arguments of the form "**keyword = value**" . We just have to identify the arguments by the parameter name.

This allows us to skip arguments or place them out of order because the interpreter is able to use the keywords provided to match the values with parameters.

Let's go back to our area function definition and make sense of this with an example:

print(area(5, 2))

10

print(area(**length = 5, width = 2))**

10

print(area(**width = 2, length = 5))**

10

The order of the arguments does not matter because the arguments are matched with the parameter names.

Note however that **keyword arguments have to be specified AFTER non-keyword arguments**.

print(area (length = 5, 2))

SyntaxError: non-keyword arg after keyword arg

No argument must receive a value more than once.

print(area (2, length = 5))

TypeError: area() got multiple values for argument 'length'

Since length is listed before width in the function definition, 2 is assumed to be the length so we end up with multiple values for length.

**Default Arguments:**

When we define a function, we can provide default values for its arguments.

When calling that function, arguments with default values are optional. If they are not provided, then the default value is used instead.

Let's define a new function definition that provides a default value for an argument. Non-default parameters must appear before default parameters in the function definition.

The function diff below is a valid function definition. **The non-default parameter, first, appears before the default parameter, second.**

def diff(first, second = 0): 参数, 默认参数=值

result = first - second

return result

Now consider the following invalid function definition. b is a non-default parameter and it appears after default parameter a. This results in a syntax error.

def invalid( a = 10, b ):

result = a - b

return result

SyntaxError: non-default argument follows default argument

Once we have specified the defaults parameters in the function definition, we can omit the corresponding arguments in the function call.

print(diff(4))

4

print(diff(first = 6))

6

print(diff(4, 2))

2

print(diff(second = 3, first = 10))

7

print(diff(second = 3))

Traceback (most recent call last):

File "C:/...", line ..., in <module>

print(diff(second = 3))

TypeError: diff() missing 1 required positional argument: 'first'

Warning:

**Do NOT use a default value for a mutable object such as a list. This results in unexpected behavior.**

Consider the following code:

def add\_one(a\_list):

**a\_list.append(1)** # append item 1 to the given list

return a\_list

first\_list = add\_one([]) # **[] denotes the empty list**

second\_list = add\_one([])

print(first\_list)

second\_list = add\_one([])

print(second\_list)

third\_list = add\_one([])

print(third\_list)

[1]

[1]

[1]

If instead of calling add\_one and specifying [] as the argument, I decide to specify a default parameter for add\_one in the function definition as follows. I would expect the same result, but I get something different.

def add\_one(a\_list = []): //使用**=**号 **对a\_list赋 default value []** 即空集

a\_list.append(1)

return a\_list

first\_list = add\_one() # use the default value

print(first\_list)

second\_list = add\_one() # use the default value

print(second\_list)

third\_list = add\_one() # use the default value

print(third\_list)

[1]

[1, 1]

[1, 1, 1]

The reason for this behavior is that **Python evaluates the default argument once first**, **when the def statement is executed** and **the same list** (**now modified**) **keeps being used**. **This may result in some hard to find bugs. 这就是为什么前面说 DO NOT USE A DEFAULT VALUE FOR L I S T AND MUTABLE Object**

To be safe, **avoid it completely** **or use the following**:

def add\_one(a\_list = None): //**在python 中 , none 相当于null 值**

**if a\_list == None:**

**a\_list = []**

**a\_list.append(1)**

**return a\_list**

Example: Print Function Arguments

The print function has optional parameters sep and end. We can use the keyword arguments to bypass the default.

sep defaults to whitespace ' '.

end defaults to new line '\n'.

If we want to specify a different separator, we use:

print ('Hello', 'World', sep = '|')

Hello|World

If we don’t want a new line between print functions calls, we use:

print('Hello', end=' ')

print('World', end=' ')

Hello World

In summary:

**Ø The positional arguments always go first, followed by any keyword arguments.**

**Ø The keywords must be chosen from the formal parameter names.**

**Ø No argument must receive a value more than once. Argument cannot receive more than one value**

**5.5. What’s in a name?**

Let's take a minute to talk about naming rules for identifiers in Python. An identifier may be a variable name, a function name, a parameter name and so on.

* Identifier names are case sensitive. grade and Grade are not the same.
* Identifiers **must begin with a letter** (a - z, A - Z) or **an underscore** ( \_ ). **Subsequent** characters can be **letters, digits or underscores**. grade, Grade1, homework\_grade and \_grade are all valid identifiers. However 1grade and grade#1 are NOT.
* An identifier cannot be a reserved word. Reserved words are words that have a special meaning in Python such as True, False and else.

The rules above are the **syntax rules**.

However for readability, we have a number of style rules in Python. The 'Style Guide for Python Code' (available from a link on the left) has the following style guidelines. We'll follow these closely in this course:

Ø Function **names should be lowercase, with words separated by underscores. Descriptive names are encouraged**.

Example: area, compute\_grade

Ø Function names typically **describe operations applied to arguments** by the interpreter (like print, add) or the name of the quantity that results (like max, sum).

Ø **Variable names should also be lowercase**, with **words separated by underscores** as necessary to improve readability.

Example: homework\_grade, final\_grade

Ø Parameter names should be lowercase, with words separated by underscores. Single-word names are preferred.

Ø Parameter names should describe the role of the parameter in the function, not just the type of value that is allowed.

Ø Single letter parameter names are acceptable when their role is obvious (s, r).

Ø **Never use "l" (lowercase ell), "O" (capital oh), or "I" (capital i) t**o avoid confusion with numerals.

**5.6. Functions and Variable Scope**

The **scope** of an identifier refers to the part of the program where that identifier is visible, where it can be accessed.

Because variables are not declared ahead of time, Python uses the location of the assignment to associate a variable with a scope. **The place where you assign a variable** in your code **determines its scope of visibility.**

* **The variables defined inside a function definition can only be seen by the code in that function definition**.  We **cannot refer to these variables from outside the function**.
* Order does matter.  We **can only access** (read) **a variable** in our scope **after it has been assigned a value**.  **The scope of a variable in a function extends from the assignment statement until the end of the function**.  It **does not include code that came before the assignment statement**.
* The **parameters defined inside a function definition can be seen by the code in that function definition def name(parameter).**  **The only other place that we can refer to them is in the function call** (as **keyword arguments**).
* Variables defined inside one function definition do not conflict with variables defined in other function definitions.  The same name may be used in different functions.

Let's take a look at a few examples to clarify scope: **def function\_name(parameters):**

def multiply(number):

    factor = 5

    result = factor \* number

**print(factor)**

    return result

answer = multiply(10)

5

The variable, factor, may be accessed from inside the function multiply.  The highlighted text represents the scope of the variable **factor**. **Factor=5 之前的任何语句无法使用 factor=5**

However if we move the print statement to outside the function, thus outside the variable scope, we get the following error:

def multiply(number):

    factor = 5

    result = factor \* number

    return result

answer = multiply(10)

**print(factor)**

NameError: name 'factor' is not defined

If we move the print statement inside the function, but before the assignment statement we also get an error.  **Order matters: the print statement is no longer in the scope.**

def multiply(number):

**print(factor)**

    factor = 5

    result = factor \* number

    return result

answer = multiply(10)

NameError: name 'factor' is not defined

We can use the keyword parameter 'number' outside the function definition:

def multiply(number):

    factor = 5

    result = factor \* number

    return result

answer = multiply(**number =** 10)

print(answer)

50

However we cannot access the parameter 'number' outside the function definition and outside the function call:

def multiply(number):

    factor = 5

    result = factor \* number

    return result

answer = multiply(10)

print(**number**)

NameError: name 'number' is not defined

The same variable name may be used in different functions.

def multiply(number):

    factor = 5

**result** = factor \* number

    return result

def half(number):

**result** = number / 2

    return result

first = multiply(10)

second = half(10)

print(first, second)

50, 5.0

There is no conflict here, the two variables have different scopes.

5.7. Pure Functions and Side Effects

**Pure functions** are supposed to take an input and return a result with **no side effects**. The function area that we defined earlier is a pure function.

By contrast, if we define the following function:

def double(n):

print(n)

result = 2 \* n

return result

double takes an input and returns a result (2\*n) but it also causes something to be printed.

Sometimes the side effect is not obvious. Consider the following function:

def special\_surface(a\_list):

"""

return the product of the first two elements of the list.

if one of the elements is 0, return the other.

"""

if a\_list[0] == 0:

a\_list[0] = 1

elif a\_list[1] == 0:

a\_list[1] = 1

result = a\_list[0] \* a\_list[1]

return result

dimensions = [10, 0, 20]

print('dimensions before the function call: ', dimensions)

answer = special\_surface(dimensions)

print(answer)

print('dimensions after the function call: ', dimensions)

The output from the lines of code above is:

dimensions before the function call: [10, 0, 20]

10

dimensions after the function call: [10, 1, 20]

The function call returns the right product, but it has changed the original list!

Note that this is mainly due to the fact that the argument is a list and lists are mutable.

To avoid the side effect, we redefine our function:

def special\_surface(a\_list):

"""

return the product of the first two elements of the list.

if one of the elements is 0, return the other.

"""

if a\_list[0] == 0:

result = a\_list[1]

elif a\_list[1] == 0:

result = a\_list[0]

else:

result = a\_list[0] \* a\_list[1]

return result

dimensions = [10, 0, 20]

print('dimensions before the function call: ', dimensions)

answer = special\_surface(dimensions)

print(answer)

print('dimensions after the function call: ', dimensions)

Now the output is:

dimensions before the function call: [10, 0, 20]

10

dimensions after the function call: [10, 0, 20]

Side effects such as this are often the source of bugs in our programs. It is best to avoid them when possible.

**5.8. The Main Function**

Now that we know how to define our own functions, we're ready to **place all our Python statements inside functions.** **We'll be following this programming practice from here on in this course.**

We'll place the main part of the code (the part that did not fall into a function but was used to call the other functions and possibly print the results) into a function that we'll call, well, main.

Going back to our area example, instead of:

def area(length, width):

""""

Compute the area of a rectangle

Parameters:

length, width (float)

Returns:

area (float)

"""

result = length \* width

return result

bedroom = area(11, 9) # first function call

kitchen = area(12, 7) # second function call

family = area(12, 12) # third function call

print(bedroom, kitchen, family)

We'll write:

def area(length, width):

""""

Compute the area of a rectangle

Parameters:

length, width (float)

Returns:

area (float)

"""

result = length \* width

return result

**def main(): //把所有不属于function的code 放到一个 main()函数里, 这些是用来call 前**

**面的函数的**

bedroom = area(11, 9) # first function call

kitchen = area(12, 7) # second function call

family = area(12, 12) # third function call

print(bedroom, kitchen, family)

I**n this case our main function takes no arguments and returns no value.**

However if you save this program in PyCharm and try to run it, you'll notice that no output is generated. **No statement is executed** either. **Function definitions don't execute any code without any function calls**. **The function calls to area are inside the main() function, however there is no function call to main(). 没有code 用来call main().**

To fix that, we add **the function call**: **main()**

def area(length, width):

""""

Compute the area of a rectangle

Parameters:

length, width (float)

Returns:

area (float)

"""

result = length \* width

return result

def main():

bedroom = area(11, 9) # first function call

kitchen = area(12, 7) # second function call

family = area(12, 12) # third function call

print(bedroom, kitchen, family)

**main()**

**5.9. Python Module**

A Python module is basically **a file that contains Python code**.

The **module name** is the **file name without the extension**.

For example when the file name is tip.py, the corresponding module name is tip.

Let's go back to our tip program and rewrite it using functions.

#-------------------------------------------------------------------------------

# Name: tip

# Purpose: tip calculator with functions

#

# Author: Rula Khayrallah

#

# Created: 09/14/2014

# Copyright: (c) Rula Khayrallah 2014

#-------------------------------------------------------------------------------

"""

Tip calculator assuming a 20% tip rate

Prompt the user for the cost of their meal.

Print the tip amount and the total cost.

"""

def get\_input():

"""

prompt the user for the cost of their meal

parameter: none

returns:the user input as a float

"""

user\_input = input('Please enter the cost of your meal in $: ')

number = float(user\_input) # convert the input string to a number

return number

def get\_tip(amount):

"""

compute the tip corresponding to the given amount

parameter: amount (float)

returns:the 20% tip rounded to 2 decimals

"""

TIP\_RATE = 20/100 # declare the tip rate constant: 20%

result = TIP\_RATE \* amount # calculate the tip amount

result = round(result, 2) # round the tip to two decimals

return result

def total(first, second):

"""

Compute the sum of two amounts rounded to 2 decimals

parameters: first, second (floats)

returns: the sum rounded to 2 decimals

"""

result = first + second

result = round(result, 2)

return result

def main():

cost = get\_input()

tip = get\_tip(cost)

print('Tip Amount: $', tip, sep = '')

pay = total(cost, tip)

print('Total amount to pay: $', pay, sep = '')

**if \_\_name\_\_ == '\_\_main\_\_': //One of the reasons for doing this is that sometimes you write**

**a module (a .py file) where it can be executed directly. Alternatively, it can also be imported and used in another module. By doing the main check, you can have that code only execute when you want to run the module as a program and not have it execute when someone just wants to import your module and call your functions themselves.**

main()

This is our module tip in the file tip.py.

The file starts with block comments that include information such as author and date.

The module itself has a docstring documenting its use.

Then come three function definitions with their own docstrings and function bodies.

And finally, we have the main part of the program which calls the various functions and uses their return values to produce some output.

Note that we have added the condition **if \_\_name\_\_ == '\_\_main\_\_':** **before** the call to **main().**

T**his is to allow the module to be imported and have its functions, classes or definitions used by some other module**.

When a module is run directly, the special variable "\_\_name\_\_" is set to "\_\_main\_\_". In this case, the main function will be executed.

**However when the module is imported** the function **main() will not be executed**. We'll see later how to import a module and use its functions in another module.

**From now on, we'll be adding if \_\_name\_\_ == '\_\_main\_\_': as shown above to all our Python files to make them valid modules that can be either run directly or imported.**

**5.10. Control Flow**

Let's take a closer look at the order in which the statements are executed in our program.

The comments and docstrings are ignored by the interpreter. So let's take them out in this section. The numbers added to the right of the statements reflect the order in which the statements are executed.

def get\_input():

user\_input = input('Please enter the cost of your meal in $: ') # 4

number = float(user\_input) # convert the input string to a number # 5

return number # 6

def get\_tip(amount):

TIP\_RATE = 20/100 # 9

result = TIP\_RATE \* amount # 10

result = round(result, 2) # 11

return result # 12

def total(first, second): # 16

result = first + second # 17

result = round(result, 2) # 18

return result # 19

def main():

cost = get\_input() # 3, 7

tip = get\_tip(cost) # 8, 13

print('Tip Amount: $', tip, sep = '') # 14

pay = total(cost, tip) # 15, 20

print('Total amount to pay: $', pay, sep = '') # 21

if \_\_name\_\_ == '\_\_main\_\_': # 1

main() # 2

The statements are executed **from top to bottom**, so the interpreter encounters **the function definitions first**. **However the statements inside the function definition are NOT executed until the function is called**.

So we get to the unindented statement: if \_\_name\_\_ == '\_\_main\_\_': **因为那些module name被设定为要经过if 条件检查, 如果符合, 运行main(), each module 不能被executed directly**

这个可以理解为**execute only if run as a script即被整体运行 而不是单个运行.**

This is the only statement in our module that is not inside a function, and that is what is executed first. When we run the program from PyCharm or from the terminal, the value of the special variable \_\_name\_\_ is set to '\_\_main\_\_' so the condition for the if statement is True and the indented statement main() is executed. *网上查阅的: (if the python interpreter is running that module (the source file)* ***as the main program****,* ***it sets the special \_\_name\_\_ variable to have a value "\_\_main****\_\_". If this file is being imported from another module, \_\_name\_\_ will be set to the module's name.)*

So we go to the function main() and execute its statements one after the other.

main() in turn calls get\_input() so we go to get\_input and execute its statements in order. When we get to the return statement of get\_input, control goes back to main and the **return value is** **saved** in the variable cost. That is why the statement cost = get\_input has two numbers 3 and 7.

Then we go on to the next statement in main(): tip = get\_tip(cost). Here get\_tip() is called with the argument cost. The statements inside get\_tip() are now executed. When we get to the return statement of get\_tip(), control goes back to main and the return value is saved in the variable tip.

After that we print the tip amount (statement 14). Then we call total and execute its statements. The return value from total is saved in the variable pay and finally we get to the last statement where we print the total amount to pay.

Note that in this simple example, **each function is called once**, and there are **no loops** so it was straightforward to assign a number to each statement. That will not be possible in more complex code.

**5.11. Global Variables? (avoid using)**

**Variables** defined **in the outer statements outside any function** are called **global variables**.

We'll examine how global variables work in Python but we'll **avoid using them in our programs because they result in code that is less readable, harder to debug and more difficult to maintain**.

Constants defined in the outer statements outside any function are called global constants. These are **OK to use as long as we only read them** and not modify them inside the functions. **Global constants are especially useful when the same constant is used in several functions.**

Example 1: Global constants are OK

"""

Tip calculator assuming a 20% tip rate

Prompt the user for the cost of their meal.

Print the tip amount and the total cost.

"""

TIP\_RATE = 20/100 # global tip rate constant: 20%

def get\_input():

"""

prompt the user for the cost of their meal

parameter: none

returns:the user input as a float

"""

user\_input = input('Please enter the cost of your meal in $: ')

number = float(user\_input) # convert the input string to a number

return number

def get\_tip(amount):

"""

compute the tip corresponding to the given amount

parameter: amount (float)

returns:the 20% tip rounded to 2 decimals

"""

result = TIP\_RATE \* amount # calculate the tip amount

result = round(result, 2) # round the tip to two decimals

return result

def total(first, second):

"""

Compute the sum of two amounts rounded to 2 decimals

parameters: first, second (floats)

returns: the sum rounded to 2 decimals

"""

result = first + second

result = round(result, 2)

return result

def main():

cost = get\_input()

tip = get\_tip(cost)

print('Tip Amount: $', tip, sep = '')

pay = total(cost, tip)

print('Total amount to pay: $', pay, sep = '')

if \_\_name\_\_ == '\_\_main\_\_':

main()

Please enter the cost of your meal in $: 20

Tip Amount: $4.0

Total amount to pay: $24.0

Example 2: Avoid global variables

count = 0 # global variable - bad practice

def count\_by\_two():

**global count**

count = count + 2

return

def main():

for i in range(5):

count\_by\_two()

print(count)

2

4

6

8

10

Note that because the variable count is assigned in the global scope (that is in the outermost code), the function main has a read access to it: print(count) in main() does not generate any error.

However in order to modify the variable count, the function count\_by\_two has to include the statement:

**global** count

That **global** statement is required to **signal that the following assignment statement is modifying a global variable** and not creating a new local variable.

Example 3: Use parameters and return values instead of global variables to communicate between functions.

def count\_by\_two(count):

result = count + 2

return result

def main():

main\_count = 0

for i in range(5):

main\_count = count\_by\_two(main\_count)

print(main\_count)

2

4

6

8

10

5.12. Lambda Functions

Anonymous functions in Python are called lambda functions.

Lambda functions offer a concise way to define functions without giving them a name. They are used to create and use functions on the fly. **Lambda functions are usually passed as arguments to other functions**. 为其他函数赋值. We'll see several examples of lambda functions use later in this course.

The syntax to specify a lambda function is as follows:

**lambda** **parameter\_name: return\_value**

The above is a Python expression. It can appear in an assignment statement or as an argument to another function.

Let's look at examples of valid lambda functions in Python:

lambda x: x \*\* 2 // x²

We can use the lambda above in the assignment statement:

**square = lambda x: x \*\* 2**

Now the variable square is a function.

>>> square(5)

25

The **above is equivalent to** the following:

**def square(x):**

**return x \*\* 2**

Here's another lambda function:

lambda word: word.lower()

We can use the lambda above in the assignment statement:

f = lambda word: word.lower()

Now the variable f is a function.

>>> f('HELLO')

'hello'

The above **is equivalent to** the following function definition:

**define f(word):**

**return word.lower()**

Here's a lambda function that takes two parameters:

lambda a, b: a + b // **lamba first, second: first + second**

**add = lambda a, b: a + b**

The **above is equivalent to** the following function definition:

**def add(a, b):**

**return a + b**

**6.1. What are they?**

We have already encountered several native data types in Python.

We've seen numeric data types such as integers, float and complex numbers. We have discussed booleans. We've also examined lists and introduced strings and range objects. These last three, lists, strings and range objects along with some other data types we'll introduce later in this course belong to a special group that is called the 'sequence data types.'

**A sequence is an ordered collection of data values.** It has an arbitrary (but finite) number of ordered elements.

The sequence data types all have similar behaviors, and they share some common properties.

|  |  |
| --- | --- |
| **6.2. Indexing** | |
|  | |
|  | |
| Sequence data types support **0 based indexing.**Let's review some examples with lists, strings and range objects.  We can use [] to access the value at a given index position:  >>> grades = [100, 85, 90]  >>> phrase = 'Python is fun!'  >>> valid\_grades = range(60, 101)  >>> grades[0]  100  >>> grades[-1]  90  >>> phrase[0]  'P'  >>> phrase[-1]  '!'  >>> valid\_grades[0]  60  >>> valid\_grades[-1]  100  We can use the method **index**to find the first appearance of a given element:  >>> grades.index(100)  0  >>> phrase.index('n')  5  >>> phrase.index('n', 6)   # **what if we start looking at position 6**? (‘what u look for’, start  position #)  12  >>> valid\_grades.index(70)  10  If the element is not found in the sequence, we get a ValueError:  >>> grades.index(75)  Traceback (most recent call last):    File "<input>", line 1, in <module>  ValueError: 75 is not in list    >>> phrase.index('a')  Traceback (most recent call last):    File "<input>", line 1, in <module>  ValueError: substring not found    >>> valid\_grades.index(50)  Traceback (most recent call last):    File "<input>", line 1, in <module>  ValueError: 50 is not in range  Sequence data types support slicing:  >>> **grades[1:3] // 注意 因为slice[start:end] 不包括最后结束点, 所以要写入end+1 .**  [85, 90]    >>> phrase[-4:-1]  'fun'    >>> valid\_grades[10:20]  range(70, 80) | |
| **6.3. Membership Test** |
|  |
|  |
| We can use '**in**' and '**not in**' to test for membership in any sequence data type:  We've already seen the membership test in lists:  >>> grades = [100, 85, 90]  >>> if 100 **in** grades:  ...     print('Excellent')  ...  Excellent  **The membership test also works with strings:**  >>> phrase = 'Python is fun!'  >>> 'e'  in phrase  False  >>> '!'  in phrase  True  And **the membership test works with range objects:**  valid\_grades = range(60, 101)  homework = int(input('Please enter the homework grade: '))  if homework **not in** valid\_grades:       print('Invalid grade entered!')    Please enter the homework grade: 30  Invalid grade entered!  Please enter the homework grade: 89 |

**6.4. Length and Count**

Sequence data types have a finite length. We can use the same function len to access it:

>>> len(grades)

3

>>> len(phrase)

14

>>> len(valid\_grades)

41

Sequence data types also provide a very handy **count method** that returns **the number of occurrences of one element**:

>>> grades.count(90)

1

>>> phrase.count('n')

2

>>> valid\_grades.count(0)

0

>>> zen = 'In the face of ambiguity, refuse the temptation to guess.'

>>> zen.count('e')

7

**6.5. Minimum and Maximum**

When the elements of a sequence data type are comparable, we can use the functions min and max to get the minimum and maximum element in that sequence.

We have already seen how to use min and max with a list whose elements are numeric:

>>> grades = [100, 85, 90]

>>> min(grades)

85

>>> temperatures = [98.4, 95.2, 90]

>>> max(temperatures)

98.4

We can also use min and max with a list whose elements are strings - strings are compared based on the alphabetical order (it's actually the unicode order).

>>> cities = ['San Francisco', 'San Jose', 'Los Angeles', 'San Diego']

>>> min(cities)

'Los Angeles'

>>> max(cities)

'San Jose'

We can also use min and max on the sequence data type string because the 'elements' of a string are characters and characters are comparable based on their alphabetical or unicode order.

>>> phrase = 'Python is fun!'

>>> max(phrase)

'y'

>>> min(phrase)

' '

Finally, the elements of a range object are integers and hence they are comparable:

>>> valid\_grades = range(60, 101)

>>> min(valid\_grades)

60

>> max(valid\_grades)

100

**6.6. More on Strings**

In addition to the common sequence operations, the string data type includes some powerful methods that come in handy in text processing.

We've seen how to go from a string to a list and back using the split() and join() methods.

We'll elaborate on the difference between methods and functions when we get to object oriented programming. For now, we'll note that the syntax to invoke a method is: object\_name.method\_name().

The method upper() returns a copy of the string with all the lowercase characters converted to uppercase. Note that you have to put the () after upper to invoke the method on the given string.

Here's how we use the method upper() to get an uppercase copy of the original string phrase:

>> phrase = 'Python is fun!'

>>> upper\_phrase = phrase.upper()

>>> print(upper\_phrase)

PYTHON IS FUN!

>> print(phrase)

Python is fun!

It is a common mistake to assume that because we have invoked upper() on a string, it has been modified to contain only uppercase characters. This is not correct. **Strings are immutable: the upper() method does not change the original string. It returns an uppercase copy.**

Similarly the method lower() returns a copy of the string with all the uppercase characters converted to lowercase.

Here's how we use the method lower() to get a lowercase copy of the original string phrase:

>>> phrase = 'Python is fun!'

>>> lower\_phrase = phrase.**lower()**

>>> print(lower\_phrase)

python is fun!

>>> print(phrase)

Python is fun!

Here again **the lower() method does not change the original string. It returns a lowercase copy**.

The **strip()** method returns a copy of the string with **the leading and trailing characters specified removed**.

Let's show how that works with some examples:

>>> zen = ' Simple is better than complex. '

**When we invoke strip without specifying an argument as in zen.strip(),** the **leading and trailing space characters** are removed.

>>> zen.strip()

'Simple is better than complex.'

Note that only leading and trailing space characters are removed. Space characters inside the string (like the one between is and better) are not removed.

**We can specify a given character, then this character is removed only if it happens to be at the beginning or end of the string:**

>>> zen.strip('.')

' Simple is better than complex. '

>>> 'Flat is better than nested.'.strip('.')

'Flat is better than nested'

We can specify more than one character to be removed:

>>> zen.strip**('. ')** # **period and white space**

'Simple is better than complex'

There are many more methods available on strings in Python. **One way to get a list of these methods is to use code completion in the interpreter shell or the editor.** **Typing the string variable name followed by a dot** brings up a list of all the methods available on the strings:

**String methods:** capitalize() casefold() center() count() encode() endswith() expantabs() find() format()

There are also some **predefined string constants** that we can conveniently use in our programs.

**To access** these constants, we need to **add the statement** **'import string'** **to the top of our program,** **after the docstring**, or if we are at the interpreter prompt, execute it from there.

**>>> import string**

>>> string.punctuation

'!"#$%&\'()\*+,-./:;<=>?@[\\]^\_`{|}~'

>>> string.digits

'0123456789‘

>>> string.ascii\_lowercase

'abcdefghijklmnopqrstuvwxyz'

>>> string.ascii\_uppercase

'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

Once we have imported string, we can use these constants for example to get rid of punctuation in a word or in a larger text:

**>>> word = 'hello??!!!'**

**>>> word.strip(string.punctuation)**

**'hello'**

Or in the case where **punctuation characters are inside the string**:

>>> phrase = 'hello??!!! (anyone there?)'

>>> **for char in string.punctuation:**

**... phrase = phrase.replace(char, ''**) # **replace punctuation with empty string**

...

>>> phrase

'hello anyone there'

**6.7. Tuples 元组**

A tuple is a **sequence data type**. It consists of **a number of values separated by commas**. Like other sequence data types, a tuple has an **arbitrary but finite length**.

>>>measurements = 5, 7, 5

>>> type(measurements)

<class 'tuple'>

>>> print(measurements)

(5, 7, 5)

The variable measurements refers to a tuple. **Tuples may or may not be enclosed in parentheses**. The variable colors below also refers to a tuple.

>>> colors = ('red', 'green', 'purple', 'black')

>>> type(colors)

<class 'tuple'>

>>> print(colors)

('red', 'green', 'purple', 'black')

Like all other sequence data types in Python, tuples are **0 indexed**. **即index 从0位开始前提是tuple不为空**

>>> colors = ('red', 'green', 'purple', 'black')

>>> colors[0]

red

>>> colors[1:3]

('green', 'purple')

We can use the **index** method to access the index of a given element:

>>> colors.**index('purple')**

2

We can use **in** and **not in** to **test for membership** in a tuple:

>>> 'yellow' in colors

False

>>> 'black' in colors

True

>>> 'green' not in colors

False

We can get the **length of a tuple**:

>>>measurements = 5, 7, 5

>>> len(measurements)

3

We can get the count of a given element in a tuple: **count() 方法 记住是用来找出现频率的**

>>> measurements.count(7)

1

>>> measurements.count(5)

2

If the tuple elements are comparable, we can get their minimum and their maximum:

>>> min(measurements)

5

>>> max(measurements)

7

If the elements of a tuple are numeric, we can also get their sum:

>>> sum(measurements)

17

**Like lists, tuple items may be of different types**.

>>> rank = ('Alice', 2**) // 注意: 定义 tuple 与定义 list 的方式相同，但整个元素集是用小**

**括 号()包围的，而不是方括号[]。**

>>> print(rank)

('Alice', 2)

**Like lists, tuples may be nested.**

>>> grades = (100, 95, 90)

>>> student = 'Bob', 20179999, grades

>>> print (student)

('Bob', 20179999, (100, 95, 90))

This is **a tuple inside a tuple**

**Tuples are immutable**. **The elements cannot be changed** **after the tuple is created… but tuples may contain lists – which are mutable**.

>>> grades = [100, 95, 90] // 这是一个list

>>> student = 'Bob', 20179999, grades // 这是一个tuple , 内嵌了上面的list

**定义 tuple = c1, c2, c3….**

>>> print (student)

('Bob', 20179999, [100, 95, 90])

**Tuples** are immutable, we **cannot change the element** at index position 0. We **get an error if we try:**

>>> student[0] = 'Alice'

Traceback (most recent call last):

File "<input>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

However**, the nested list inside the tuple is mutable**:

>>> student**[2][1]=**0 // student tuple 内的第二位是list, list内的第一位 换为0

>>> print(student)

('Bob', 20179999, [100, 0, 90])

Even though we can’t modify the elements of a tuple**, we can always make the variable reference a new tuple**.

>>> rank = ('Alice', 2) // **用变量rank来储存元祖**

>>> rank[1] //**用 tuple名字[index位置] 来命名元组内的任何元素**

2

We cannot change the element at index position 1, we get an error if we try:

>>> rank[1] = 1

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

**But we can reassign the variable rank to a different tuple:**

>>> rank = ('Alice', 1) // **通过重新赋值来改变tuple**

>>> rank[1]

1

We often **use tuples to assign multiple values at once**.

>>> colors=('red', 'green', 'yellow')

>>> x, y, z = colors // **用字符 (逗号分隔) 来对应/储存tuple内每个元素**

>>> x

'red'

>>> y

'green'

>>> z

'yellow '

>>> width, length, height = 5, 7, 5

>>> width

5

>>> length

7

>>> height

5

This is known as **unpacking a tuple**.

There are two special tuples: the **empty tuple** and the **singleton tuple** (**tuple with a single element**):

>>> **empty=()**

>>> print(empty)

()

**>>> type (empty)**

**<class 'tuple'>**

**To create a tuple with a single element, we need to add a trailing comma , after the element:**

>>> **single = 'alice',**

>>> **another\_single = (5,)**

>>> type (single)

<class 'tuple'>

>>> print(single)

('alice',)

>>> print(another\_single)

(5,)

Without the comma, the interpreter will assign the string 'alice' to single and the integer 5 to another\_single.

**6.8. Tuple or List?**

**Tuples** look a lot like lists except that **they have parentheses** instead of square brackets. So how do we decide to use one rather than the other?

There are **two things to keep in mind when making that choice**:

**Tuples are immutable, lists are mutable.**

**Tuples are faster than lists.**

**If you are dealing with constant data, tuples make your code safer**. **There is no way to accidentally change a tuple.**

Let's say for example that I need **a data structure** (that **DO NOT CHANGE** once created) for the days of the week. That data is constant. There will be no need to change it. A tuple is a good choice here.

days\_of\_week = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')

On the other hand if I need a data structure to store my friends' names and I am a fairly sociable and outgoing person, I'll choose a list since I can see myself making new friendships or possibly dropping some old ones in the future:

friends = ['Amy', 'Ben', 'Carol', 'Dan']

That being said, the choice does not have to be final: we can easily go from tuples to lists and back:

**We can convert a tuple to a list**, **using the function list().** We say that **list()** **thaws a tuple**. 被称作 **解冻** tuple

>>> days\_of\_week = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')

>>> list(days\_of\_week)

['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

We can also convert a list to a tuple, using the function **tuple().** We say that tuple() **freezes a list**. **冷冻** list

>>> friends = ['Amy', 'Ben', 'Carol', 'Dan']

>>> tuple(friends)

('Amy', 'Ben', 'Carol', 'Dan')

**ALL SEQUENCE TYPES: list, tuple, range**

**7.1. What is a set? 无序不重复元素集**

A set is an **unordered** **collection of unique items**. We can think of it as a bag.

**The items don't need to have the same type** but **they** need to be immutable. **集合内元素不可变**

We **create a set by listing its elements** between curly braces **{}**:

>>> pronouns\_sub = {'I', 'you', 'he', 'she', 'it', 'we', 'they'} //建集合: **变量名={‘元1’, ‘元素2’,…}**

>>> type(pronouns\_sub)

<class 'set'>

The **items of a set are unique** – **even if we enter the same item more than once**. **Duplicates are not added. 重复输入相同的元素, 不会被加入到集合内, 所以输入和没输入一样.**

>>> my\_set = {**4**, 6, **4**, 'blue'}

>>> my\_set

{**4**, 6, 'blue'}

***Sets are unordered:***

>>> pronouns\_sub = {'I','you','he','she','it','we','they'}

>>> pronouns\_sub

{'we', 'it', 'she', 'you', 'they', 'he', 'I'}

**The order I used to enter the set pronouns\_sub was NOT preserved**.

**The items of a set must be immutable** (or more precisely hashable—stay the same for lifetime). **Sets may not include a list as an item.**

>>> invalid\_set = {1, 'green', [1, 4, 6]}

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

TypeError: **unhashable type: 'list**'

**To create an empty set, we have to use set():**

**>>> empty = set()**

>>> type (empty)

<class 'set'>

**Do NOT use {} to create an empty set. That will create an empty dictionary instead**.

**7.2. Adding and Removing Items**

We can add items to a set:

>>> sizes = {14, 6, 4, 8}

>>> sizes.add(2)

>>> sizes

{8, 2, 4, 6, 14}

If we try to add a value that already exists in the set, it will do nothing. It won’t raise an error.

>>> sizes.add(6)

>>> sizes

{8, 2, 4, 6, 14}

We can take items out of a set using discard():

>>> sizes.discard(2)

>>> sizes

{8, 4, 6, 14}

>>> sizes.discard(10)

>>> sizes

{8, 4, 6, 14}

discard() does not nothing if the item does not exist.

We can also take items out of a set using remove():

>>> sizes.remove(6)

>>> sizes

{8, 4, 14}

>>> sizes.remove(16)

Traceback (most recent call last):

File "<input>", line 1, in <module>

KeyError: 16

remove() raises an exception if the item does not exist.

We can also **use pop() to remove and return an arbitrary 任意的item from the set**.

>>> sizes.pop() // **POP() 删除随机选择的任一集合内元素**

8

>>> sizes

{4, 14}

>>> sizes.pop()

4

>>> sizes

{14}

>>> sizes.pop()

14

>>> sizes

set()

>>> sizes.pop()

Traceback (most recent call last):

File "<input>", line 1, in <module>

KeyError: 'pop from an empty set'

**pop() raises an exception if the set is empty.**

We can use clear() to remove all items from the set.

>>> colors = {'red', 'white' , 'blue'}

>>> colors.clear()

>>> colors

set()

**7.3. Set Operations**

We can get the intersection of two sets:

A & B -> the set of items that are in A and in B

This is equivalent to A.intersection(B)

>>> pronouns\_sub = {'I','you','he','she','it','we','they'}

>>> pronouns\_obj = {'me','you','her','him','it','us','them'}

>>> pronouns\_sub & pronouns\_obj

{'you', 'it'}

We can get the difference of two sets:

A – B -> **the set of items that are in A but not in B**

This is equivalent to A.difference(B)

>>> pronouns\_sub - pronouns\_obj

{'he', 'they', 'she', 'I', 'we'}

We can get the union of two sets:

A | B -> the set of items that are in either A or in B

This is equivalent to A.union(B)

>>> pronouns\_sub | pronouns\_obj

{'they', 'it', 'she', 'me', 'we', 'her', 'I', 'he', 'them', 'you', 'us', 'him'}

We can get the symmetric difference of two sets:

A ^ B -> the set of items that are in either A or in B but not in both

This is equivalent to A.symmetric\_difference(B)

>>> pronouns\_obj ^ pronouns\_sub

{'me', 'him', 'he', 'we', 'them', 'I', 'her', 'they', 'us', 'she'}

We can also test for subsets:

A <= B -> A is a subset of B

**All the items in A are also in B**

This is equivalent to **A.issubset(B)**

>>> pronouns\_sub = {'I','you','he','she','it','we','they'}

>>> pronouns\_obj = {'me','you','her','him','it','us','them'}

>>> pronouns\_obj <= pronouns\_sub

False

>>> {'I', 'you'} <= pronouns\_sub

True

We can test for supersets:

A >= B -> A is a superset of B

All the items in B are also in A

This is equivalent to A.issuperset(B)

>>> pronouns\_sub = {'I','you','he','she','it','we','they'}

>>> pronouns\_obj = {'me','you','her','him','it','us','them'}

>>> pronouns\_obj >= pronouns\_sub

False

>>> pronouns\_obj >= {‘her’, ‘us’}

True

**7.4. What else can we do with a set?**

We can test for **membership in a set**:

>>> pronouns\_sub = {'I','you','he','she','it','we','they'}

>>> pronouns\_obj = {'me','you','her','him','it','us','them'}

**>>> 'me' in pronouns\_sub**

False

>>> 'me' in pronouns\_obj

True

Familiar functions we have seen with other data types work with sets too:

>>> sizes = {14, 6, 4, 8}

>>> **len(sizes**)

4

When the elements of a set are comparable, we can get their minimum, their maximum and we can sort them:

>>> **min**(sizes)

4

>>> **max**(sizes)

14

The function sorted allows us to **sort a set into a list**:

>>> **sorted(sizes)**

[4, 6, 8, 14]

If the elements of a set are numeric, we can also get their sum:

>>> **sum(sizes)**

32

**7.5. Why sets?**

So when do we use a set instead of a list or a tuple?

**Membership testing is faster with sets.**

We usually use sets **when the order is not important** **and** **when we are are dealing with unique items**.

It is also convenient sometimes to convert other data structures to a set **in order to count the number of unique items or to get rid of duplicates**.

**We can convert any sequence to a set by using the set() function.**

We can **create a set out of a list**:

>>> grades = [95, 100, 85, 75, 95, 60, 100]

>>> unique\_grades = **set(grades)**

>>> unique\_grades

**{75, 100, 85, 60, 95}**

>>> len(unique\_grades)

5

Note that **95 and 100 were included only once** and **the order of the list was NOT preserved** in the set.

So to **count the number of unique items** in the list grades, we can simply write

>>> **len(set(grades))**

5

To **get rid of the duplicate items**, we **go from a list to a set and back to a list**:

>>> **list(set(grades))**

[75, 100, 85, 60, 95]

Similarly, we can **create a set out of a string**:

>>> some\_text = 'find the unique characters‘

>>> len(some\_text)

26

**To find the unique characters in the string,** we write:

>>> **set(some\_text)**

{'q', 's', 'r', 'u', 't', 'a', ' ', 'c', 'e', 'd', 'f', 'i', 'h', 'n'}

**To count the number of unique characters** in a string:

>>> **len(set(some\_text))**

14

We can also **create a set out of a tuple**:

>>> choices = ('continue', 'red', 'green', 'yellow', 'continue', 'cancel')

>>> **set(choices)**

{'green', 'red', 'continue', 'yellow', 'cancel'}

We can even **create a set out of a range** object:

>>> a\_grades = range(90, 101)

**>>> set(a\_grades)**

{96, 97, 98, 99, 100, 90, 91, 92, 93, 94, 95}

**8.1. Dictionaries, Keys and Values**

**Mutables: list, dict, array, set**

**Immutables: number, string, tuple, frozen set, bytes,**

A dictionary is an **unordered** collection of key: **value pairs**.

**Dictionary\_name ={‘key’:’value’, ‘key’:’value’}**

Each entry contains an index/key and a value separated by a colon.

In a dictionary, the indices are called **keys.**

**Dictionary key must be immutable(string, number, immutable tuple), but values can be mutable(list).**

**The purpose of a dictionary is to store and retrieve values that are indexed by descriptive keys.**

We create a dictionary by specifying the key:value pairs between {}:

>>> address\_book={'Dan': '555-5678', 'Alice' :'555-1234'}

In this case the keys are strings. **Dictionary keys** can be of any **immutable type 不可变类型** (strings, numbers, **immutable tuples** but not lists.)

**Keys are unique within a dictionary.**

**Dictionaries are unordered.**

>>> address\_book={'Dan': '555-5678', 'Alice' :‘555-1234'}

>>> type(address\_book)

<class 'dict'>

>>> address\_book

{'Alice': '555-1234', 'Dan': '555-5678'}

The entries are not in the same order we defined.

There is no reason to care about the order, since the elements of a dictionary are never indexed with integer indices. Instead, we use the keys to look up the corresponding values.

We can access the value corresponding to a certain key with the square brackets:

>>> address\_book={'Dan': '555-5678', 'Alice' :'555-1234'}

>>> **address\_book['Dan'] // to call a dictionary, use dictionary\_name[‘key’]**

'555-5678'

>>> address\_book['Alice']

'555-1234'

**Python dictionaries are optimized for retrieving the value when we know the key,** but not the other way around.

We can also get the value corresponding to a certain key with the **get()** method: **get. 用来取key对应的值**

>>> address\_book={'Dan': '555-5678', 'Alice' : '555-1234'}

>>> **address\_book.get('Dan‘)**

'555-5678'

>>> address\_book.get('Alice‘)

'555-1234'

**8.2. Membership Test**

When we attempt to retrieve a dictionary key that does not exist, we get an error:

>>> address\_book={'Dan': '555-5678', 'Alice' :'555-1234'}

>>>print( address\_book['Bob'])

Traceback (most recent call last):

File "<input>", line 1, in <module>

KeyError: 'Bob'

To avoid that, we can check for membership in a dictionary, using in:

>>> 'Charlie' in address\_book

False

**>>> 'Alice' in address\_book**

True

We should always check if a key exists in a dictionary before attempting to retrieve its value.

>>> if 'Bob' in address\_book:

... print(address\_book['Bob'])

...

>>> if 'Dan' in address\_book:

... print(address\_book['Dan'])

...

555-5678

**IMPORTANT NOTE: THERE IS NO ADVANTAGE to use LAMBA function,** it makes the code less readable. **TO DEFIND SEQUENCE OF FUNCTION, put the first to run first. E.G:**

**lower\_case\_list = text.lower().split() // Remember this.**

**8.3. Updating a dictionary**

A dictionary is a **mutable data type**. IT is not sequence data type, because dict is unordered. We can add items to a dictionary, modify existing items and delete items from a dictionary.

Adding Items:

We can add key value pairs to the dictionary by assigning a value to a new key:

>>> address\_book={'Dan': '555-5678', 'Alice' :'555-1234'}

>>> **address\_book['Jim']= '555-8899'**

>>> address\_book

{'Alice': '555-1234', 'Dan': '555-5678', 'Jim': '555-8899'}

Note that no error is generated here even though the key 'Jim' did not initially exist in the dictionary. That's because we are assigning a value to the key 'Jim', not trying to retrieve it.

Modifying Items:

A dictionary can have at most one value for each key.

Assigning a value to an existing dictionary key replaces the old value with the new one.

>>> address\_book

{'Alice': '555-1234', 'Dan': '555-5678', 'Jim': '555-8899'}

>>> address\_book['Alice']

'555-1234'

>>> address\_book['Alice']= '999-3333'

>>> address\_book

{'Alice': ‘999-3333', 'Dan': '555-5678', 'Jim': '555-8899'}

Deleting Items:

We can use **del** to delete an item given its key: the key and its corresponding value will be deleted. Del dict\_name[key]

>>> address\_book

{'Alice': '999-3333', 'Dan': '555-5678', 'Jim': '555-8899'}

>>> del address\_book['Dan']

>>> address\_book

{'Alice': '999-3333', 'Jim': '555-8899'}

If the key does not exist, we get an error:

>>> del address\_book['Bob']

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

KeyError: 'Bob'

We can also use pop to delete an entry from the dictionary. Pop also returns the corresponding value.

>>> address\_book

{'Alice': '999-3333', 'Dan': '555-5678', 'Jim': '555-8899'}

>>> address\_book.pop('Dan')

'555-5678'

>>> address\_book

{'Alice': '999-3333', 'Jim': '555-8899'}

If the key does not exist and we don’t specify a default, we also get an error:

>>> address\_book.pop('Bob')

Traceback (most recent call last):

KeyError: 'Bob'

We can specify a default value to be returned if the key does not exist.

>>> address\_book.pop('Bob', '999-9999')

‘999-9999'

>>> address\_book.pop('Bob', None)

No error is generated in this case.

**8.4. Iterating over a dictionary**

We can use a **for ... in loop** to iterate over a dictionary.

**Iterating over a dictionary is equivalent to iterating over its keys:**

>>> address\_book = {'Alice' :'555-1234', 'Dan': '555-5678'}

>>> **for friend in address\_book:**

... print(friend)

...

Dan

Alice

**To print the phone number associated with each friend**, we write:

>>> for friend in address\_book:

... print(friend, **address\_book[friend]) // remember to access a key’s value use this--dictionaryname[key name]**

...

Dan 555-5678

Alice 555-1234

Note since the items in the dictionary are not ordered, the friends' names printed above are not sorted. Sometimes it is useful **to iterate over a dictionary in some order**. To do that, we can call use the function **sorted()** **on the dictionary first**. The function returns a sorted list of the dictionary keys and we can then iterate over this sorted list.

Let's see how to do that for our example above. We start with:

>>> address\_book = {'Alice' :'555-1234', 'Dan': '555-5678'}

We **get a sorted list of the dictionary keys first**:

>>> sorted\_friends\_list = sorted(address\_book)

>>> sorted\_friends\_list

['Alice', 'Dan']

We then iterate over the sorted list. We can still access the corresponding value in the dictionary as address\_book[friend]:

>>> for friend in sorted\_friends\_list:

... print(friend, address\_book[friend])

...

Alice 555-1234

Dan 555-5678

We'll often skip assigning a variable to the sorted list and write the above in the more compact form:

>>> for friend in sorted(address\_book):

... print(friend, address\_book[friend])

...

Alice 555-1234

Dan 555-5678

**8.5. What else can we do with a dictionary?**

We can use the function len to get the number of items in a dictionary:

>>> address\_book = {'Alice' :'555-1234', 'Dan': '555-5678'}

>>> len(address\_book)

2

If the keys of a dictionary are comparable, we can use max and min to get their maximum and minimum:

>>> max(address\_book)

'Dan'

>>> min(address\_book)

'Alice'

We can use the method **values()** to get the current values in the dictionary:

>>> address\_book.values()

dict\_values(['555-5678', '555-1234'])

The method **values()** returns a 'view' in Python 3. In Python 2, it used to return a list.

Unlike a list, which is static, a view is a dynamic or 'live' object. This means that when the dictionary changes, the view reflects these changes.

>>> address\_book = {'Alice' :'555-1234', 'Dan': '555-5678'}

>>> my\_view = address\_book.values()

>>> my\_view

dict\_values(['555-5678', '555-1234'])

Now when we change one value in the dictionary, the view reflects that change.

>>> address\_book['Dan'] = '555-5555'

>>> my\_view

dict\_values(['555-5555', '555-1234'])

Going back to our original dictionary, we can iterate over the values, just like a list:

>>> address\_book = {'Alice' :'555-1234', 'Dan': '555-5678'}

>>> for phone\_number in address\_book.values():

... print(phone\_number)

555-5678

555-1234

We can also **call the sorted()** function on the values:

>>> **sorted(address\_book.values())**

['555-1234', '555-5678']

It is sometimes useful to get the values in a list, or a set (to count unique values for example).

>>> list(address\_book.values())

['555-5678', '555-1234']

>>> set(address\_book.values())

{'555-5678', '555-1234'}

Similarly, we can use the keys() method to get all the keys in the dictionary:

>>> address\_book.keys()

dict\_keys(['Dan', 'Alice'])

The method keys() also returns a 'view' in Python 3. In Python 2, it used to return a list.

Finally we can use the items() method to get the current items (as a tuple) in the dictionary:

>>> address\_book.items()

dict\_items([('Dan', '555-5678'), ('Alice', '555-1234')])

The method items() also returns a view in Python 3.

We can then iterate over the items. Each iteration gives us a tuple:

>>> for my\_tuple in address\_book.items():

... print(my\_tuple)

...

('Dan', '555-5678')

('Alice', '555-1234')

Or we can **unpack each tuple into a name and a phone variables** as follows:

>>> for (name, phone) in address\_book.items():

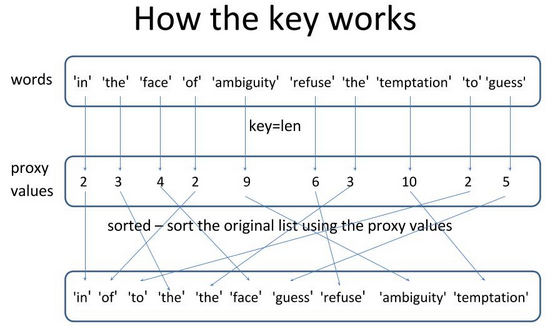
... print(name, phone)

...

Dan 555-5678

Alice 555-1234

|  |
| --- |
| **8.6. Sorting Fun** |
|  |
|  |
| Let's take a closer look at the built in function sorted(). It will be very useful when dealing with dictionaries.  **The Basics:**  sorted() takes as input an **iterable** and returns a **list. Sorted(iterable object, key, reverse)**  Some of the **iterables** we have seen so far are **strings, lists, tuples, sets and dictionaries.**  Of course the items in the iterable have to be comparable.  Strings are sorted in ascending alphabetical order.  sorted() may be called on a list:  >>> sorted('hello')  ['e', 'h', 'l', 'l', 'o']  sorted() may be called on a tuple:  >>> sorted (('blue', 'red', 'green'))  ['blue', 'green', 'red']  sorted() may be called on a set:  >>> sorted ({'I', 'you', 'it'})  ['I', 'it', 'you']  As we've seen earlier, **when sorted() is  called on a dictionary,  it operates on the dictionary keys not on values if not defined.**  >>> sorted({'Jim': '555-8899', 'Alice': '999-3333'})  ['Alice', 'Jim']  When the items in the iterable are numbers, they are sorted in ascending numeric order:  >>> sorted ([8, 2, 3])  [2, 3, 8]  **sorted()** also **takes two optional keyword arguments**, **key and reverse**.  **Reverse Sort:**  The **reverse argument defaults to False**. **If it is set to True,  the items are sorted in reverse order.**  >>> sorted('hello', **reverse = True**)  ['o', 'l', 'l', 'h', 'e']  >>> sorted ([8, 2, 3], **reverse = True**)  [8, 3, 2]  >>> sorted (('blue', 'red', 'green'), reverse = True)  ['red', 'green', 'blue']  >>> sorted ({'I', 'you', 'it'}, reverse = True)  ['you', 'it', 'I']  >>> sorted({'Jim': '555-8899', 'Alice': '999-3333'}, reverse = True)  ['Jim', 'Alice']  words = ['in', 'the', 'face', 'of', 'ambiguity', 'refuse', 'the', 'temptation', 'to', 'guess']  >>> sorted(words)  ['ambiguity', 'face', 'guess', 'in', 'of', 'refuse', 'temptation', 'the', 'the', 'to']  sorted(words, reverse = True)  ['to', 'the', 'the', 'temptation', 'refuse', 'of', 'in', 'guess', 'face', 'ambiguity']  **Sorting with a key function:**  The optional key argument allows us to **specify a function** that **is used to compute proxy values for each item.** **The sort is then performed using these proxy values**.  **如果使用key定义,那么sort会使用ke键定义的代理值, 这里的key不是dictionary key本身.**  **The specified key function may be a built-in function such as len or a user defined function**.  Note that **this key argument is NOT** related to the dictionary key.  Let's make sense of this with some examples:  >>> words = [‘in', 'the', 'face', 'of', 'ambiguity', 'refuse', 'the', 'temptation', 'to', 'guess']  To sort the words in the list words from shortest to longest, we can write:  >>> sorted(words, **key = len**)  ['in', 'of', 'to', 'the', 'the', 'face', 'guess', 'refuse', 'ambiguity', 'temptation'] |



Here the len() function is used to **compute a proxy value for each item in the original list** words.  **These proxy values are then used to determine the order of the items in the output list**.

**Sorting a dictionary by value:**

The key argument can be used to **sort a dictionary by value instead of by key.**

Suppose we have created a dictionary 'my\_count' that keeps track of how many times a given letter appears in a string.

For the string 'the cat in the hat', my\_count would look like:

>>> my\_count = {'c': 1, 'a': 2, 'e': 2, 't': 4, 'i': 1, 'h': 3, 'n': 1}

To print the letters in alphabetical order, we can write:

>>> print(sorted(my\_count))

['a', 'c', 'e', 'h', 'i', 'n', 't']

But what if we want to print these **letters sorted by their frequency**?

What if we used values()?

>>> my\_count.values()

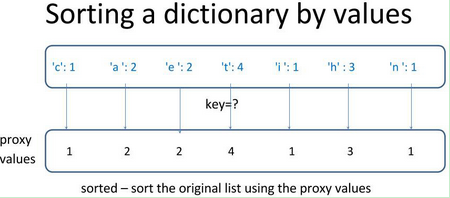
dict\_values([1, 2, 2, 4, 1, 3, 1])

>>> sorted(my\_count.values())

[1, 1, 1, 2, 2, 3, 4]

This gives us the values sorted in increasing order.  However we need to have the letters corresponding to these values.

Our next option is to use sorted() with the key argument:

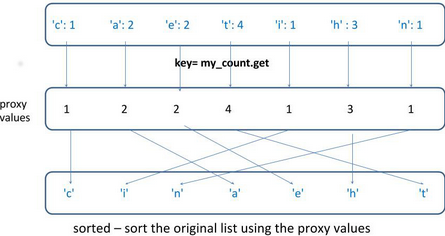


We would like the proxy values to be the letter counts.

**The question is how do we define our key argument function so that we end up with these proxy values?**

We need the key argument function to return the value associated with the letter in the count dictionary.  We can use the dictionary method get():

my\_count.get(letter) returns the value associated with letter in the dictionary.  That's our letter count.



>>> my\_count = {'c': 1, 'a': 2, 'e': 2, 't': 4, 'i': 1, 'h': 3, 'n': 1}

>>> sorted(my\_count, **key = my\_count.get**)

['c', 'n', 'i', 'e', 'a', 'h', 't']

Or to get the letter in decreasing order of frequency:

>>> sorted(my\_count, **key = my\_count.get, reverse = True**)

['t', 'h', 'e', 'a', 'c', 'n', 'i']

To print these letters sorted by decreasing frequency, we can iterate over the list generated by sorted as follows:

>>> sorted\_letters = sorted(my\_count, key=my\_count.get, reverse = True)

>>> for letter in sorted\_letters:

        print (letter + ':', my\_count[letter])

t: 4

h: 3

e: 2

a: 2

c: 1

n: 1

i: 1

We'll often see the above written more concisely as:

for letter in sorted(my\_count, key = my\_count.get, reverse = True):

        print (letter + ':', my\_count[letter])

t: 4

h: 3

e: 2

a: 2

c: 1

n: 1

i: 1

**It is common to specify a lambda function expression as the value of the key argument.**

>>> sorted(my\_count,**key = lambda letter: my\_count[letter]**)

['c', 'i', 'n', 'a', 'e', 'h', 't']

The anonymous lambda function here takes a letter as its argument and returns the corresponding count. The above is equivalent to the following:

def **get\_count**(letter):

    return my\_count[letter]

sorted(my\_count,**key = get\_count**)

**max and min with a key:**

The max and min functions also take an optional key argument that works in a similar manner:

>>> words = ['in', 'the', 'face', 'of', 'ambiguity', 'refuse', 'the', 'temptation', 'to', 'guess']

>>> sorted(words, key = len)

['in', 'of', 'to', 'the', 'the', 'face', 'guess', 'refuse', 'ambiguity', 'temptation']

>>> **max**(words, **key = len**)

'temptation'

>>> **min** (words, **key = len**)

'in'

Going back to our letter count example, we can get the letters with the highest and lowest frequencies:

>>> my\_count = {'c': 1, 'a': 2, 'e': 2, 't': 4, 'i': 1, 'h': 3, 'n': 1}

>>> max(my\_count, **key = my\_count.get**)

't'

Or using a lambda function expression:

>>> min(my\_count, **key = lambda letter: my\_count[letter]**)

'n'

**8.7. A Dictionary Example**

Our goal in this example is to investigate letter distributions in English.

We’ll start by picking some representative text: the lyrics to the Beatles song Yesterday.

Since we have not yet seen how to read from a text file in Python, let's use a string variable for our song lyrics:

song = '''

Yesterday, all my troubles seemed so far away

Now it looks as though they're here to stay

Oh, I believe in yesterday

Suddenly, I'm not half the man I used to be

There's a shadow hanging over me.

Oh, yesterday came suddenly

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

'''

So we need to **count the letters in the song**. Let’s define a function count\_letters to do that.

First note that the song has upper and lower case letters. We want to count them as one letter. Y and y are the same. We'll use the string method lower() to get a lower case version of the text.

There is also a number of punctuation characters . , and ' that we don’t care about counting. We don’t want to count white space either. So we'll **import the string module** to get access to the string **constant** **string.ascii\_lowercase**. We'll only count the letters in there.

We have already seen how to use the count() method on strings to count the number of occurrences of a given character. We'll use **count()** in our function.

We'll **create a dictionary entry for each letter** and **assign a value** to it.

Here's our function definition:

import string

def count\_letters(text):

"""

count the letters found in the string text

Parameters: text, a string

Returns: a dictionary with items of the form letter: count

"""

letter\_count={} # initialize the dictionary **Do NOT use {} to create an empty set. That will create an empty dictionary instead**. SO {} here create an empty dictionary

lower\_text = text.lower() # convert the text to lower case

# no need to count punctuation characters - go over the lower case letters

for letter in string.ascii\_lowercase:

if letter in lower\_text:

**letter\_count[letter] = lower\_text.count(letter) # save the count**

return letter\_count # return the dictionary

Now in our main program, we'll include the following steps:

1. Call count\_letters to get the song\_count dictionary.

2. Based on the song\_count dictionary, print the total number of letters that appear at least once in the song.

Based on the song\_count dictionary, print those letters sorted alphabetically, with their corresponding count.

Based on the song\_count dictionary, print the letters sorted by count with the highest count letters first.

def main():

song = '''

Yesterday, all my troubles seemed so far away

Now it looks as though they're here to stay

Oh, I believe in yesterday

Suddenly, I'm not half the man I used to be

There's a shadow hanging over me.

Oh, yesterday came suddenly

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

'''

song\_count = count\_letters(song)

print('The song contains', len(song\_count), 'letters')

print("The letters sorted alphabetically:")

for letter in sorted(song\_count): # print letters in alphabetical order

print(letter + ':', song\_count[letter])

# build the list of letters sorted by decreasing frequency

frequency\_list = sorted(song\_count, key = **song\_count.get**, reverse = True)

print("The letters sorted by frequency:")

for letter in frequency\_list:

print(letter + ':', song\_count[letter])

Note that we used the **two keyword parameters of the sorted() function: key and reverse**.

**sorted(song\_count, key = song\_count.get, reverse = True)** gives us a list of the dictionary **keys (letters)** sorted by the **proxy value that is their corresponding count value**. We **save that list in frequency list**.

We then iterate over this list to print it.

Putting it all together, here's our program:

# -------------------------------------------------------------------------------

# Name: lettercount

# Purpose: demonstrate the use of dictionaries

#

# Author: Rula Khayrallah

#

# Created: 10/06/2013

#-------------------------------------------------------------------------------

"""

Count the letters in the Beatles song 'Yesterday'

Print all the letters found in the song alphabetically

followed by the number of times they appear in the song.

Print all the letters found in the song in descending order of frequency.

"""

import string

def count\_letters(text):

"""

count the letters found in the string text

Parameters: text, a string

Returns: a dictionary with items of the form letter: count

"""

letter\_count = {} # initialize the dictionary

lower\_text = text.lower() # convert the text to lower case

# no need to count punctuation characters - go over the lower case letters

for letter in string.ascii\_lowercase:

if letter in lower\_text:

letter\_count[letter] = lower\_text.count(letter)

return letter\_count # return the dictionary

def main():

song = '''

Yesterday, all my troubles seemed so far away

Now it looks as though they're here to stay

Oh, I believe in yesterday

Suddenly, I'm not half the man I used to be

There's a shadow hanging over me.

Oh, yesterday came suddenly

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

Why she had to go I don't know she wouldn't say

I said something wrong, now I long for yesterday

Yesterday, love was such an easy game to play

Now I need a place to hide away

Oh, I believe in yesterday

'''

song\_count = count\_letters(song)

print('The song contains', len(song\_count), 'letters')

print("The letters sorted alphabetically:")

for letter in sorted(song\_count): # print letters in alphabetical order

print(letter + ':', song\_count[letter])

# build the list of letters sorted by decreasing frequency

frequency\_list = sorted(song\_count, key = song\_count.get, reverse = True)

print("The letters sorted by frequency:")

for letter in frequency\_list:

print(letter + ':', song\_count[letter])

if \_\_name\_\_ == '\_\_main\_\_':

main()

And here's the corresponding output

The song contains 22 letters

The letters sorted alphabetically:

a: 45

b: 5

c: 5

d: 28

e: 65

f: 4

g: 13

h: 27

i: 27

k: 3

l: 20

m: 10

n: 30

o: 43

p: 4

r: 19

s: 36

t: 31

u: 9

v: 6

w: 19

y: 34

The letters sorted by frequency:

e: 65

a: 45

o: 43

s: 36

y: 34

t: 31

n: 30

d: 28

h: 27

i: 27

l: 20

w: 19

r: 19

g: 13

m: 10

u: 9

v: 6

b: 5

c: 5

p: 4

f: 4

k: 3

**9.1. Opening a file**

A file is a collection of data stored in one unit, identified by a file name.

We may have:

a text file: spider.txt

a program file: python.exe

an iTune song: 02 Clocks.m4a

a picture: GrumpyCat.jpg …

In Python, before we can read from or write to a file, we need to **open** it. We use the built in **function open()** to do that. The general syntax to open a file is as follows:

**some\_file = open(filename, mode, encoding)**

The open function returns a **file handle.** We assign it to the variable some\_file. some\_file is just an arbitrary variable name. We can use it from there on to refer to the file.

So, for example we can write:

my\_file = open('C:/Users/Rula/Documents/CS21A/spider.txt')

**Absolute and Relative Paths:**

The file name above includes an absolute path: 'C:/Users/Rula/Documents/CS21A/spider.txt'.

Note that we used **forward slashes**/. In Python, forward slashes always work, even on Windows.

We can also use a relative path as in:

my\_file = open('spider.txt')

The relative path will start from our working directory.

To **figure out what our working directory** is, we can use the **os module** as follows:

**import os**

**print(os.getcwd())**

On windows, I get:

C:\Users\Rula\Documents\CS21A

On a Mac OS:

/Users/rulakhayrallah/PycharmProjects/CS21A

We’ll talk more about the os module later in the course.

**It is best to use relative path names whenever possible to ensure the portability of our programs.**

**Mode:**

mode is an **optional argument**. It defaults to **'rt' which means open for reading in text mode**. So we can write:

my\_file = open('spider.txt', 'r')

The mode can be set to any of the following.

**'r' open for reading (default).**

'w' open for writing, overwriting the existing file.

'x' create a new file for writing – generate an error if the file already exists.

'a' open for writing, appending to the end of the file if it exists.

'b' binary mode.

't' text mode (default)

'+' open a file for reading and writing ('r+' or 'w+')

We can combine modes when it makes sense as in 'rt' (read in text mode) or 'w+b' (read/write in binary).

**Encoding**

Encoding is also an **optional** argument. It is **only applicable in text** mode. Encoding **determines how the sequence of Unicode characters** (text) is mapped onto bytes.

my\_file = open('spider.txt', 'r', **encoding='utf-8'**)

The default encoding is platform dependent, but any encoding supported by Python can be passed.

CP-1252 is a common encoding on computers running Microsoft Windows.

UTF-8 is used on Unix and Mac OS.

**It is best to set the encoding to utf-8 when possible to make sure that our programs are platform independent.**

**9.2. Closing a file**

When we’re done with a file, we should close it. To do that, we use the close() method as follows:

my\_file = open('spider.txt', 'r', encoding='utf-8')

my\_file.**close()**

We can **test if the file is closed** by checking **the value of 'closed'**, **a boolean attribute** defined on the file object. closed indicates the state of the file. It is True if the file is closed and False otherwise.

>>> my\_file = open('spider.txt')

>>> my\_file.closed

False

>>> my\_file.close()

>>> my\_file.closed

True

So typically, we open the file, do something with it then close it.

some\_file.open(filename,…)

Do something with the file

some\_file.close()

**If something happens while we are working with the file, that file could stay open for much longer than necessary.**

In the next section, we'll introduce a way to get around this problem.

**9.3. A better way - the with statement**

**The with statement offers us a safer way to open and close files.**

**with** open('spider.txt', 'r', encoding='utf-8') **as** my\_file:

do something with the file my\_file

Let's take note of the following:

* Here again, **my\_file** is an arbitrary variable name. We can use it from there on to refer to the file. (It is **the file handle**). It is as if we had: **my\_file = open('spider.txt') 相同作用**
* The with statement starts a code block, like an if statement or a for loop. Note the colon at the end of the with statement. The code block after the with statement is indented.
* We get to do whatever we need to do with my\_file inside the indented code block.
* **When the with block ends, Python calls my\_file.close() automatically**.**自动关闭**

**Advantage:**

**No matter how or when we exit the with block, Python will close that file**… even if our code crashes.

In this course, we'll **always use the with statement** when **working with files**.

**9.4. Reading the whole file**

The text file spider.txt is available under Resources. Please download it or copy it to the working directory on your system so that you can follow along.

We can read the whole file at once. We use the **read() method** to do that:

with open('spider.txt', 'r', encoding='utf-8') as my\_file:

text = my\_file.read()

print (text)

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.

Here we are reading the whole file into a variable, text – text is a string.

Note that the print statement is outside the 'with' indented block. At this point, the file is closed. The content of the file has been saved in the variable text and we can access it in our program even though the file has been closed.

We can add some more statements to manipulate or extract some specific data from the file we just read: we'll use the lower() string method to get a copy of the text we just read in lower case and then print it. We'll also use the count() string method to get the number of times 'spider' or 'spout' appeared in the text we just read.

with open('spider.txt', 'r', encoding='utf-8') as my\_file:

text = my\_file.read()

print (text)

lower\_text = text.lower()

print (lower\_text) # print a lower case version of the song

print ('spider ->', lower\_text.count('spider')) # print some statistics

print ('spout ->', lower\_text.count('spout'))

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.

the itsy bitsy spider went up the water spout.

down came the rain, and washed the spider out.

out came the sun, and dried up all the rain,

and the itsy bitsy spider went up the spout again.

spider -> 3

spout -> 2

Note that **reading the whole file at once with the read() method is only feasible if the file is small enough.**

**9.5. Reading one line at a time**

We can read the file one line at a time as follows:

with open('spider.txt', 'r', encoding = 'utf-8') as my\_file:

**for line in my\_file: // 这是for…in… loop**

**print(line)**

Here again, line is an arbitrary variable name.

We can think of a text file as a sequence of items where each item is a line. We can iterate over the sequence with a for... in loop. When we do, we get one line at a time. So it is helpful (but not necessary) to call our iteration variable line so we know what kind of entity we are dealing with.

with open('spider.txt', 'r', encoding = 'utf-8') as my\_file:

**for line in my\_file:**

**print(line)**

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.

We end up with extra lines in the output because **each line read from the file (except the last one) has a new line character included in it**. Then print appends a new line each time it is called.

**To fix that, we can specify the end parameter for print. This way it will NOT default to new line**.

with open('spider.txt', 'r', encoding = 'utf-8') as my\_file:

**for line in my\_file:**

**print(line, end = '')**

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.

**9.6. One word at a time?**

Often, our goal is to get to the individual words in the file, to analyze them or extract some information from them. We can do that by using the string method **split()** on each line. The split() method **splits a string into a list of strings** **based on some delimiter**. The **default delimiter is white space**, but it could be anything we specify.

with open('spider.txt', 'r', encoding = 'utf-8') as my\_file:

for line in my\_file:

for **word** in **line.split():**

print(**word**)

The

Itsy

Bitsy

Spider

went

up

the

water

spout.

Down

came

the

rain,

and

washed

the

spider

out.

Out

…(skipped)

Note that since split determined the boundaries of words using white space, not punctuation, **'spout.' and 'rain,' were identified as words**.

**To fix that**, we can **use** the string method **strip()** and the constant string.punctuation as follows:

**import string**

with open('spider.txt', 'r', encoding = 'utf-8') as my\_file:

for line in my\_file:

for word in line.split():

word = word.**strip**(**string.punctuation**) # **get rid of punctuation**

print(word)

The

Itsy

Bitsy

Spider

went

up

the

water

spout

Down

came

the

rain

and

washed

the

spider

out

Out

...

Note that the variable names **my\_file, line and word** are **completely arbitrary任意**. We could have used x, y and z but then we would have quickly lost track of what we’re dealing with.

It is best to use representative variable names so we know what entity we’re working with.

**9.7. Writing to a file**

Now that we know how to read a file, let's see how to write to it.

We can **append text at the end of the file**. To do that we specify **'a'** for the mode. // **a 是 mode append text at the end**

with open('spider.txt', **'a'**, encoding = 'utf-8') as my\_file:

my\_file**.write**('And again and again.')

The **write()** method **writes a string to the file**. Note that **the argument** specified to write **has to be a string**. If we are writing a number, we need to convert it to a string first.

spider.txt now contains the following:

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.And again and again.

**The write() method does not start a new line.** If we need a new line, we add the new line character \n to the string we are writing: \n 换行符

with open('spider.txt', 'a', encoding = 'utf-8') as my\_file:

**my\_file.write('\nAnd again and again.')**

spider.txt now contains the following:

The Itsy Bitsy Spider went up the water spout.

Down came the rain, and washed the spider out.

Out came the sun, and dried up all the rain,

And the Itsy Bitsy Spider went up the spout again.And again and again.

And again and again.

We can also **overwrite an existing file** by using the **'w' mode**:

with open('spider.txt', 'w', encoding = 'utf-8') as my\_file:

my\_file.write('Bye Bye Spider')

spider.txt now contains the following:

Bye Bye Spider

Note that **you don’t have to actually write something** in 'w' mode to overwrite the file.

**Just opening it in 'w' mode will overwrite it**.

with open('spider.txt', 'w', encoding = 'utf-8') as my\_file:

pass

spider.txt is now empty.

**The pass statement does nothing**. **It is simply used as a placeholder** because the syntax of the with statement requires an indented block.

**Both the append mode 'a' and the write mode 'w' will create the file if it does not already exist:**

with open('newspider.txt', 'w', encoding = 'utf-8') as my\_file:

my\_file.write('This is a brand new spider')

with open('newestspider.txt', 'a', encoding = 'utf-8') as my\_file:

my\_file.write('This is a brand new spider')

newspider.txt and newestspider.txt are created in the working directory and contain the following:

This is a brand new spider

9.8. A Spell Checker Example

Our task today is to build a rudimentary spell-checker.

We’ll 'learn' the words by reading a very large text file and identifying the words in it. The words in that reference file are assumed to be spelled correctly. Then we'll read an input file and identify the words in it as well. Any word in the input file that is not found in the reference file is flagged as a potential misspelled word.

It looks like we have three major subtasks here:

'Learn' the words by reading a very large text file and identifying the words in it.

Read an input file and identify the words in it.

Flag any word in the input file that is not found in the reference file as a potential misspelled word.

What data structure do we use to keep track of all the words we encounter? List, tuple, set, dictionary?

A set seems like a good choice. We only need to store a word once, even if it appears multiple times in a given file. We don’t need to keep the words in any order: we’ll just need to check if a word exists or not. Sets are an efficient choice for testing membership.

The first and second subtask are very similar. We'll be reading a file and identifying the words in it. In the first case it is the reference file and in the second it is the input file. We'll implement one generic function that does that. We learned in module 9.6 how to do that:

import string

def get\_words(filename):

"""

Get the words from a file

Parameter: file name

Return: a set containing all the words in that file

"""

words\_set = set()

with open(filename, 'r', encoding='utf-8') as file:

for line in file:

line = line.lower() # convert the line to lower case

for word in line.split(): # then split it into words

# take out leading and trailing special characters

word = word.strip(string.punctuation + string.digits)

if word: # if word is not an empty string

words\_set.add(word) # add the word to our set

return words\_set

Here are a few observations about the code above:

We always use the with construct to open and close files.

filename is the name of the file on our computer and file is the variable that we'll use to refer to that file inside my program.

Since the file is very large, we read it line by line. When we iterate over a file with a for loop, we get one line in each iteration.

We use lower() to convert the line to lower case.

Remember that line.split() returns a list of the words in the string, line, using white space as the delimiter string.

We iterate over this list with a for loop, expecting one word in each iteration.

We take out leading and trailing punctuation characters. We also take out leading and trailing numbers. This may result in an empty string if the initial word only contained punctuation characters or numbers so we make sure that the resulting word is not the empty string (if word:) before adding it to our set.

Our third subtask can be achieved by taking the difference of two sets: the set of words in the input file - the set of words in the reference file - these are the misspelled words.

We can then write our program as follows:

#-------------------------------------------------------------------------------

# Name: spellcheck

# Purpose: Demonstrate files and the use of sets

#

# Author: Rula Khayrallah

#

# Created: 9/30/2014

#-------------------------------------------------------------------------------

"""

A rudimentary spell checker

Prompt the user to enter a reference file name and an input file name.

This can be any text file in any language.

The program detects spelling errors in the input file name

and prints them out.

The program determines that a word is an error if it has not been seen

it in the reference.

"""

import string

def get\_words(filename):

"""

Get the words from a file

Parameter: file name

Return: a set containing all the words in that file

"""

words\_set = set()

with open(filename, 'r', encoding='utf-8') as file:

for line in file:

line = line.lower() # convert the line to lower case

for word in line.split(): # then split it into words

# take out leading and trailing special characters

word = word.strip(string.punctuation + string.digits)

**if word: # if word is not an empty string**

**words\_set.add(word) # add the word to our set**

**return words\_set**

def main():

ref\_name = input('Please enter the reference file name: ')

reference = get\_words(ref\_name) # get the reference words

input\_name = input('Please enter the input file name: ')

input\_words = get\_words(input\_name) # get the input words

misspellings = input\_words - reference # get the misspelled words

if misspellings: # if the set is not empty

print ('The following words may have been misspelled:')

for word in misspellings:

print (word)

if \_\_name\_\_ == '\_\_main\_\_':

main()

We can test our spell checker at this point:

Using 'Pride and Prejudice' by Jane Austen (1813) as our reference and a recent technology news article, we get the following:

The following words may have been misspelled:

developers

storage

push

notifications

apps

google

mobile

backend

starter

bitcoin

ios

code

queries

blog

We can even test with text in different languages:

Using 'Les Miserables' by Victor Hugo as our French language reference and a French children song – spelled correctly, we get the following:

The following words may have been misspelled:

Plumerai

**Assignment #5 :**

**out\_file.write(word+":"+str(reference[word])+'\n') 只需把dictionary中的key 对应的值转为string 类型 后面 加换行符**

**max(count\_dict, key = len ) 要返回最长的key, 只需对dictionary 使用 Max 方法**

10. Object Oriented Programming

10.1. When and Why?

Object oriented programming started in the Artificial Intelligence community in the 1960s. However the term 'object oriented programming' was not introduced until the 1970s.

In the late 1980s and 1990s it became the main programming paradigm used in the creation of new software**. It was well suited to handle the rapidly increasing size and complexity of software.**

Object-oriented features have been added to many previously existing languages.

**Python was designed to support object-oriented programming** while still compatible with functional methodology.

**10.2. What's an object?**

So far we have been talking about functions and data: functions perform some operations and return data and data is operated upon.

An object, on the other hand is an abstraction that combines information and behavior. We can think of it as a building block with some desired functionality. The functionality is encapsulated in the object: we know the 'what' not the 'how'.

Let's take a look at some object examples next.

An object example: a car:

A car has some **attributes:** make, model, mileage, fuel efficiency, etc…

But it also has a certain **behavior**.

When we drive the car, we use up some gas.

When we drive the car, the car mileage increases.

At some point, we may run out of gas.

We may fill up the tank with gas, and so on…

Let’s try to organize these attributes and behavior:

**Attributes:**

make

model

mileage

gas in tank

fuel efficiency

**Behavior:**

drive (6 miles)

add gas(7.6 gallons)

tank empty? ->

get the current mileage ->

Another object example: a bank account

Attributes:

owner : 'Alex'

balance: $20

Behavior:

deposit ($100)

withdraw($20)

get the balance ->

One last object example: a person

Attributes:

name

birthday

profile picture

friends

Behavior:

add a friend(person)

unfriend(person)

get the list of friends ->

**OOP Terminology:**

The attributes correspond to the fields, properties or instance variables.

The behavior corresponds to the methods.

In OOP, we say that **methods are invoked on a particular object.**

**In Python everything is an object.**

**Integers, strings, lists, dictionaries are objects.**  **Data types, functions and modules are also objects.**

And of course, we'll create our own objects.

**10.3. Classes**

**A class is a group of objects with common behavior.**  The class definition contains a template for the objects.

Defining a class in Python is simple. We define the class and start coding.

**class** Car(object): // **class** **Classname**(**type** of class e.g object):

"""

Represent a car in my virtual world.

"""

A Python class starts with the reserved word **class, followed by the class name**.

The header indicates that the new class is a Car, which is a kind of object. object is a built-in type.

**Class names are usually capitalized and use CapWords**.

Just like function definitions and compound statements, the class definition header ends with a colon.

**Everything in a class definition is indented**. **The first line not indented is outside the class**.

We can define methods inside a class definition, we'll see how to do that in the upcoming sections.

The **docstring** explains what the class is for. The docstrings conventions document (PEP 257 - available from the left navigation tab) recommends **inserting a blank line before and after all docstrings that document a class.**

Later on, we’ll come back to this class docstring and add more details to describe the Car object.

The class is like a factory for creating objects. To create a Car object, we call Car as if it were a function.

**my\_car = Car(…)**

Creating a new object is called **instantiation实例化**, and the object is an instance of the class.

The above statement creates a new instance of the class Car and assigns this object to the local variable my\_car.

**There is no explicit new operator like in other languages.**

Some classes inherit from other classes. Car just inherits from the object built-in class. We’ll cover inheritance a little later.

There is nothing that a Python class absolutely must have, other than a name.

\_\_init\_\_():

**The method that initializes objects** has a special name**, \_\_init\_\_** .

Think of it as **the constructor for the class** – even though the object has already been created. Python中的\_init\_ 等同于 constructor in other language.

**It is not required**, but if it is there it is called immediately after an instance of the class is created.

By convention, the \_\_init\_\_() method is the first method defined for the class.

class Car(object):

"""

Represent a car in my virtual world.

"""

def \_\_init\_\_(self, make, model, fuel\_efficiency, mileage = 0, gas = 0): // 可以给argument default 值 比如 0

**The first argument of every class method**, including the \_\_init\_\_() method, is **always a reference to the current instance of the class.**

By convention, this argument is named **self**. (**Other languages use 'this' instead.)**

self is **not a reserved word** in Python. It’s a naming convention.

We need to specify self explicitly when defining the method. However we do not specify it when calling the method. Python will add it for us automatically.

Where do the rest of \_\_init\_\_’s arguments come from?

**Remember that to create a Car object, we call Car as if it were a function.** This is where we pass the arguments that will be used by \_\_init\_\_.

my\_car = Car('Honda', 'Civic', 28)

Now is a good time to go back to the Car class docstring and update it to reflect the arguments that \_\_init\_\_ needs.

class Car(object):

"""

Represent a car in my virtual world.

Arguments:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float, optional): current mileage on car in miles, defaults to 0.

gas (float, optional): current gas in the tank in gallons, defaults to 0.

"""

**10. Object Oriented Programming**

**10.4. Instances**

**To instantiate a class, we simply call the class as if it were a function,** passing the arguments that the \_\_init\_\_() method requires.

The return value will be the **newly created object**. We call that the **new instance**.

my\_car = Car('Honda','Civic', 28, 12000, 3.5)

When we call the print function on an instance, Python usually tells us what class it belongs to.

print(my\_car)

<\_\_main\_\_.Car object at 0x02D34450>

**10.5. Instance Variables**

What does an object need to remember?

• Car: gas, mileage, make, model, fuel efficiency

• Account: holder, balance

**Instance variables are the attributes** that are specific to the instance. Each object (or instance) has a separate copy of the instance variable.

Let’s go back and fill in the code for our \_\_init\_\_ method in the Car class:

class Car(object):

"""

Represent a car in my virtual world.

Arguments:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float, optional): current mileage on car in miles, defaults to 0.

gas (float, optional): current gas in the tank in gallons, defaults to 0.

"""

def \_\_init\_\_(self, make, model, fuel\_efficiency, mileage = 0, gas = 0):

self.make = make //self 指向Car object

self.model = model

self.fuel\_efficiency = fuel\_efficiency

self.mileage = mileage

self.gas\_in\_tank = gas

In Python, we **use the dot** notation **to designate** an **attribute (or instance variable)** of an object.

self.make, self.model, self.fuel\_efficiency, self.mileage and self.gas\_in\_tank are all **attributes** of the object.

self.make, self.model, self.fuel\_efficiency, self.mileage and self.gas\_in\_tank are also called **instance variables**.

They are **global** to the instance. That means that we can access them from other methods.

self.make, self.model, self.fuel\_efficiency, self.mileage and self.gas\_in\_tank are separate from the parameters make, model, fuel\_efficiency , mileage and gas. The names don’t have to be the same.

We can go back and expand our class docstring to document these instance variables.

class Car(object):

"""

Represent a car in my virtual world.

Arguments:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float, optional): current mileage on car in miles, defaults to 0.

gas (float, optional): current gas in the tank in gallons, defaults to 0.

Attributes:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float): current mileage on the car in miles.

gas\_in\_tank (float): current gas in the tank in gallons.

"""

def \_\_init\_\_(self, make, model, fuel\_efficiency, mileage = 0, gas = 0):

self.make = make

self.model = model

self.fuel\_efficiency = fuel\_efficiency

self.mileage = mileage

self.gas\_in\_tank = gas

**Instance variables are specific to one instance of a class**. For example, if we create two Car instances with different make and model, they will each remember their own make and model.

**A little demo:**

>>> from car import Car

My class definition for the class Car is saved in a module named car.py.

We have seen how to import the whole module before: import car.

However that would mean every time I want to call Car from the interpreter or from outside the module, I have to prefix with the module name:

my\_car= car.Car('Honda', 'Civic', 28)

When I use from car import Car, I can just use Car without prefixing it with the module name:

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

Here I have just created a new instance of the Car class. I also have assigned that new instance to the variable my\_car.

>>> your\_car = Car('Porsche', '911', 23)

… and another instance of the Car class assigned this time to the variable your\_car.

>>> my\_car.make

'Honda'

>>> your\_car.make

'Porsche'

These are 2 different instance variables, associated with 2 different objects.

my\_car and your\_car are 2 different instances, 2 different objects

Each instance will remember its own make, its own model and its own mileage.

>>> my\_car.model

'Civic'

>>> your\_car.model

'911'

>>> my\_car.fuel\_efficiency

28

>>> your\_car.fuel\_efficiency

23

**10.6. Identity**

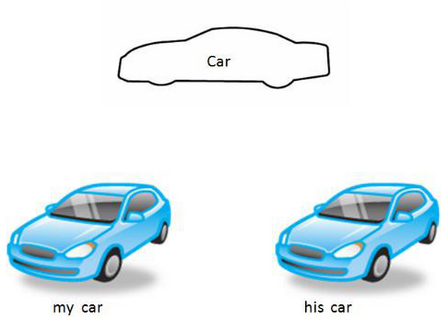
Each new car instance has its own attributes (make, model, etc.) and the value of these attributes is **independent of** other objects of the same class.

**To enforce** this separation**, every object has a unique identity**.

Object identity is compared using the is and is not operators.

>>> my\_car = Car('Honda', 'Civic', 28)

>>> his\_car = Car('Honda', 'Civic', 28)



>>> my\_car is his\_car

Even though they are constructed from identical calls, the objects my\_car and his\_car **are not the same.** Two separate objects have been created here.

Note that in English, I would say that he and I have the same car: same make, same model, maybe even same color. **But that does not mean the same physical car!** We each have our own car, **we do not share a car.**

**Identity and Assignment:**

Now consider the following:

>>> his\_car = Car('Honda', 'Civic', 28)

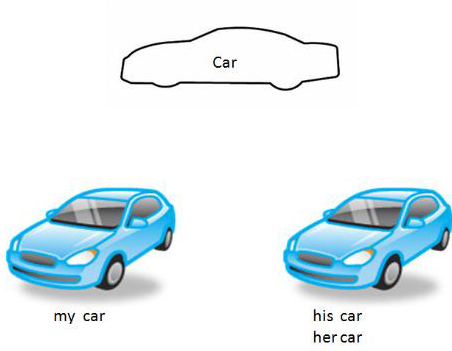
>>> her\_car = his\_car // **= 任何情况下都是赋值!**

>>> her\_car is his\_car

True

As we’ve seen before with lists, an **assignment does not create a new object**. **his\_car and her\_car are pointing to the same object in memory. Use same instance. 同一个目标**

**New objects are only created when a class is instantiated with Car(…).**



>>> his\_car.fuel\_efficiency

28

>>> her\_car.fuel\_efficiency

28

>>> his\_car.fuel\_efficiency = 22

>>> her\_car.fuel\_efficiency

22

Because his\_car and her\_car are pointing to the same object, changing his\_car.fuel\_efficiency changes her\_car.fuel\_efficiency.

**10.7. Methods**

Methods describe what we do with the objects.

The bank account:

• deposit

• withdraw

The car:

• drive

• add gas

Methods are **basically functions** that **operate on the object** or **perform object-specific computations.**

We say that **methods are invoked on a particular object**.

Methods are **defined inside a class definition** in order to make the relationship between the class and the method explicit.

**The syntax for invoking a method** is different from the syntax for calling a function.

**Defining a method:**

We define a method with **the def** **keyword** just as we would define a function.

def tank\_empty(self):

"""Return True if the gas tank is empty."""

return self.gas\_in\_tank == 0

A method has a docstring just like a function.

That docstring is available when we get help on the class Car:

>>> help(Car)

Help on class Car in module car:

class Car(builtins.object)

| Represent a car in my virtual world.

|

| Arguments:

| …

| Attributes:

| …

| Methods defined here:

|

| \_\_init\_\_(self, make, model, fuel\_efficiency, mileage=0, gas=0)

|

| tank\_empty(self)

| Return True if the gas tank is empty.

We can also get help on the method directly, we just prefix it with the class name.

>>> help(**Car**.tank\_empty)

Help on function tank\_empty in module car:

**tank\_empty(self)**

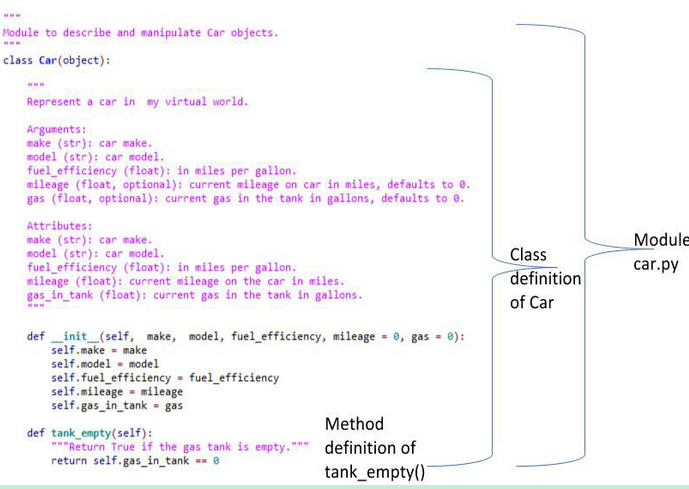
**Return True if the gas tank is empty.**

Each method definition includes the special first parameter self. Self refers to the object on which the method is invoked. In our example self will point to my\_car or your\_car or his\_car.

So **to access an attribute** of that particular object, we use **self.attribute\_name** inside the method.

**All methods have access to the object** via the **self parameter**, and so they can all access and manipulate the object's state.

And this is how everything fits together:



**Invoking a method:**

The syntax for invoking a method is different from the syntax for calling a function.

The method **is invoked on an object and the self parameter does not have to be specified.**

From the interpreter, we can write:

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> **my\_car.tank\_empty()**

True

Or we can add a main() function inside the module car.py:

def main():

my\_car = Car('Honda','Civic', 28, 12000, 3.5)

**if my\_car.tank\_empty():**

print ("You're out of gas!")

else:

print('Ready to go!')

if \_\_name\_\_ == '\_\_main\_\_':

main()

We can also invoke the method like a function (not OOP!), but then we lose the object oriented aspect and we have to provide the parameter. We also have to prefix the method name with the class in this case.

>>> my\_car = Car('Honda', 'Civic', 28)

>>> **Car.tank\_empty(my\_car)**

True

**10.8. More methods for our car**

Now we can go ahead and define more methods for our class. Remember that we are trying to **capture the behavior of the car and encapsulate it in the object**.

**Behavior: Adding gas to the car**

We'll start by writing the docstring for our method add\_gas():

"""

Add gas to the car.

Parameter:

amount (float): the amount of gas to be added in gallons.

Returns:

the updated car object (Car).

"""

Now we can write the method definition. We know that **the first parameter to any class method is self**. The 'amount' parameter will be listed after self.

def add\_gas(self, amount):

"""

Add gas to the car.

Parameter:

amount (float): the amount of gas to be added in gallons.

Returns:

the updated car object (Car).

"""

**self.gas\_in\_tank += amount**

**return self**

gas\_in\_tank is an attribute of our car objects (an instance variable). To access the gas\_in\_tank attribute of a given instance inside the method, we use self.gas\_in\_tank.

Note that the add\_gas **method returns the updated car object, self**. When we write methods that return updated objects, **we can chain method invocations so that the object is operated on by several methods sequentially.**

For instance, we can **invoke the tank\_empty() method on the car after adding gas to it** as follow:

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> my\_car.tank\_empty()

True

>>> my\_car.add\_gas(10).tank\_empty()

False

Here we are **invoking tank\_empty() on the return value from my\_car.add\_gas(10).**

We'll see more examples of **method chaining** in this course.

**Behavior: Driving the car**

We drive the car a given distance. We'll define the requirements on our drive() method interface as follows:

Parameter:

distance (float): the distance to be driven in miles.

Returns:

the updated car object (Car).

The method definition will then start with:

def drive(self, distance):

Here again, **the first parameter to the drive() method is self**. The distance parameter is listed after self.

Before writing the method definition, let’s work through an example.

Suppose we have 3 gallons of gas in the tank.

Our fuel efficiency is 25 miles per gallon.

If we invoke drive on our car:

my\_car.drive(25) – > we expect success

The mileage on the car needs to be updated:

mileage = mileage + 25

mileage = mileage + distance

The gas in the tank needs to be updated.

We use up one gallon to drive 25 miles, so:

gas in tank = gas in tank – 1

gas in tank = gas in tank – (distance / fuel efficiency )

Now suppose that we want to drive another 100 miles.

At this point we have 2 gallons of gas in the tank.

Our fuel efficiency is still 25 miles per gallon.

If we invoke drive on our car:

my\_car.drive(100) – > we expect to run out of gas

That’s because with 2 gallons in the tank, the maximum distance we can drive is 25 \* 2 = 50 miles

maximum distance = fuel efficiency \* gas in tank

So we expect the car to go for 50 miles and then stop because it’s out of gas.

The mileage on the car needs to be updated:

mileage = mileage + maximum distance

The gas in the tank needs to be updated: the tank is empty at this point.

gas in tank = 0

Now we can write our method definition as follows.

def drive(self, distance):

"""

Drive a car a given distance, if possible.

If there is not enough gas, drive as much as possible.

Parameter:

distance (float): the distance to be driven in miles.

Returns:

the updated car object (Car)

"""

max\_distance = self.fuel\_efficiency \* self.gas\_in\_tank

if distance <= max\_distance: # we can drive the distance

self.mileage += distance

self.gas\_in\_tank = self.gas\_in\_tank - distance /self.fuel\_efficiency

else: # not enough gas

self.mileage += max\_distance # drive as far as possible

self.gas\_in\_tank = 0

return self

And this is how everything fits together now in the module car.py:

class Car(object):

"""

Represent a car in my virtual world.

Arguments:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float, optional): current mileage on car in miles, defaults to 0.

gas (float, optional): current gas in the tank in gallons, defaults to 0.

Attributes:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float): current mileage on the car in miles.

gas\_in\_tank (float): current gas in the tank in gallons.

"""

def \_\_init\_\_(self, make, model, fuel\_efficiency, mileage = 0, gas = 0):

self.make = make

self.model = model

self.fuel\_efficiency = fuel\_efficiency

self.mileage = mileage

self.gas\_in\_tank = gas

def tank\_empty(self):

"""Return True if the gas tank is empty."""

return self.gas\_in\_tank == 0

def add\_gas(self, amount):

"""

Add gas to the car.

Parameter:

amount (float): the amount of gas to be added in gallons.

Returns:

the updated car object (Car).

"""

self.gas\_in\_tank += amount

return self

def drive(self, distance):

"""

Drive a car a given distance, if possible.

If there is not enough gas, drive as much as possible.

Parameter:

distance (float): the distance to be driven in miles.

Returns:

the updated car object (Car)

"""

max\_distance = self.fuel\_efficiency \* self.gas\_in\_tank

if distance <= max\_distance: # we can drive the distance

self.mileage += distance

self.gas\_in\_tank = self.gas\_in\_tank - distance /self.fuel\_efficiency

else: # not enough gas

self.mileage += max\_distance # drive as far as possible

self.gas\_in\_tank = 0

return self

**10.9. Test Drive**

Let's take a car on a test drive to demonstrate the behavior of our Car class. Please follow along in the Python console. Just make sure you have copied and saved the module car.py first.

We first import the class Car.

>>> from car import Car

We'll pick a Porsche 911 with a 22 miles per gallon fuel efficiency.

>>> test\_car = Car('Porshe', '911', 22)

We'll fill it up with 15 gallons of gas.

>>> test\_car.add\_gas(15)

<car.Car object at 0x0000000003C1CCF8>

Let's take it on a 100 miles drive.

>>> test\_car.drive(100)

<car.Car object at 0x0000000003C1CCF8>

What's the mileage now?

>>> test\_car.mileage

100

Are we out of gas?

>>> test\_car.tank\_empty()

False

Let's go on a 500 miles drive.

>>> test\_car.drive(500)

<car.Car object at 0x0000000003C1CCF8>

Are we out of gas?

>>> test\_car.tank\_empty()

True

Oops. What's the mileage now?

>>> test\_car.mileage

330.0

Because the methods add\_gas() and drive() return the updated object, we can use method chaining as follows:

>>> test\_car.add\_gas(15).drive(30).drive(80)

<car.Car object at 0x0000000003BFCCF8>

>>> test\_car.mileage

440.0

Here we added 15 gallons of gas, drive 30 miles and then another 80 miles with the following statement:

test\_car.add\_gas(15).drive(30).drive(80)

**10.10. Class Variables**

Some attribute values are **shared across all objects of a given class.** Such attributes are **associated with the class itself,** rather than any individual instance of the class.

These **attributes** are also called **class variables**.

For instance, let us say that for all our cars in the Car class, we want to specify the distance unit (miles). We want to remember this distance unit. **However it is the same for all the cars in our class.**

**Class variables are created by assignment statements in the class definition**, **outside of any method definition.**

class Car(object):

"""

Represent a car in my virtual world.

Arguments:

make (string): car make.

model (string): car model.

fuel\_efficiency (float): in miles per gallon.

mileage (float, optional): current mileage on car in miles, defaults to 0.

gas (float, optional): current gas in the tank in gallons, defaults to 0.

"""

distance\_unit = 'miles'

def \_\_init\_\_(self, make, model, fuel\_efficiency, mileage = 0, gas = 0):

self.make = make

self.model = model

self.fuel\_efficiency = fuel\_efficiency

self.mileage = mileage

self.gas\_in\_tank = gas

**distance\_unit is a class variable of the class Car. It is available before creating any instances of the class.**

**Class attributes are available through any instance of the class:**

>>> from car import Car

>>> my\_car = Car('Honda','Civic', 25)

>>> print(my\_car.distance\_unit)

miles

They are also available through a direct reference to the class:

>>>print(Car.distance\_unit)

miles

Class attributes can be used as **class-level constants**, but they are not really constants. we can also change them. If we want to change them, **we’ll have to do so through the reference to the class**, not through an instance reference.)

Let's see how that works:

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> your\_car = Car('Porsche', '911', 23)

>>> my\_car.make

'Honda'

>>> your\_car.make

'Porsche'

>>> Car.distance\_unit

'miles'

>>> my\_car.distance\_unit

'miles'

>>> your\_car.distance\_unit

'miles'

We have just created **two instances** of the Car class, my\_car and your\_car and we can access the class vaiable through either of them.

Now let's say we want to **change the distance unit to km**, and we first write:

>>> **my\_car.distance\_unit = 'km'**

>>> my\_car.distance\_unit

'km'

**Here we have created a new instance variable, distance\_unit.** **We have not changed the class variable distance\_unit.**

>>> your\_car.distance\_unit

'miles'

>>> Car.distance\_unit

'miles'.

**If we want to change the class variable, we’ll have to do so through the reference to the class:**

>>> **Car.distance\_unit = 'km'**

>>> your\_car.distance\_unit

'km'

**10.11. Private Instance Variables? 私有实例变量**

Private instance variables that cannot be accessed except from inside an object **don’t exist in Python.**

We don’t need getter and setter methods to access instance variables.

Not only can we can access (read) the car mileage, make, etc… from outside the class and from the interpreter prompt, we can also modify these instance variables from outside the class:

>>> from car import Car

>>> my\_car = Car('Honda','Civic', 25)

>>> print(my\_car.mileage)

0

>>>my\_car.mileage = 6000

>>> print(my\_car.mileage)

6000

However, there is a **convention惯例**that is followed by most Python code: a name prefixed with an underscore (for example **\_private**) **有下划线前缀的通常是API中非公开部分 私有** should be treated as a non-public part of the API (whether it is a function, a method or an instance variable). It should be considered an implementation detail and subject to change without notice.

**10.12. Why OOP?**

Object Oriented Programming allows us to tame the **size** of **complexity** of software by dividing it into logical and semi independent entities.

It makes the maintenance of this software much easier to handle.

**The interface become separate from the implementation**. The interface has to keep track of the '**what**' not the '**how**'. For objects, the methods a class provides should not depend on how the attributes are represented or manipulated.

A method with a given interface may be implemented in several ways.

As a result, when we design the interface carefully, we can change the implementation when the need arises without changing the interface.

**11.1. What are Magic Methods?**

Magic methods, also called special methods, are methods that we define to add magic to our classes.

Their name is surrounded by double underscores ( like \_\_init\_\_ or \_\_str\_\_).

Remember that in Python, **anything that starts with \_ is supposed to be private to the class and not part of the interface.** The magic methods perform their magic behind the scenes **but they are not supposed to be explicitly invoked from outside the class.**

We have already seen the most basic magic method, \_\_init\_\_.

\_\_init\_\_ allows us to define the initialization behavior of an object.

Even though it is NOT invoked explicitly, \_\_init\_\_ gets magically invoked and passed whatever parameters are specified in the object instantiation.

new\_car = Car(‘Honda’, ‘Civic’, 28)

In the next sections, we'll examine some other useful magic methods in Python.

**11.2. The \_\_str\_\_ Method**

The \_\_str\_\_ method is another magic method that we can define in our classes.

We define it to return a string representation of an object.

When we print an object, Python invokes the \_\_str\_\_ method on that object to figure out what to print.

It is very useful for debugging.

Let’s go back to our Car class definition.

We did not have a \_\_str\_\_ method defined there. What happens when we try to print a Car object?

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> print(my\_car)

<car.Car object at 0x02AB6CD0>

Now let’s add a \_\_str\_\_ method definition to our Car class.

It goes right after \_\_init\_\_.

We can choose to return whatever we want to represent a car.

Let’s just return the make and the model.

def \_\_str\_\_(self):

return self.make + ' ' + self.model

Now what happens when we print a Car object?

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> print(my\_car)

Honda Civic

The \_\_str\_\_ method is also invoked when we call the str() function on an object.

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> str(my\_car)

'Honda Civic'

**11.3. Operator Overloading**

Changing the behavior of an operator so that it works with user-defined classes is called **operator overloading**. **操作符重载.**

Python's magic methods provide a simple way to **make objects behave like built-in types**. They **allow us to specify the behavior of operators** (+, -, ==, <, [], etc…) on any object.

We can then use these operators on our own classes as if they were built-in types.

For every operator in Python there is a corresponding special method.

Let's go back to our car example and illustrate why it is convenient to have such methods.

>>> my\_car = Car('Honda', 'Civic', 28)

>>> his\_car = Car('Honda' , 'Civic', 28)

>>> my\_car is his\_car

False

We have seen that my\_car and his\_car are two different objects.

However we still may want to capture the fact that these cars have the same make and model.

We could write a method in the Car class as follows:

def same\_car(self, other):

"returns True if both cars have the same make and model"

result = self.make == other.make and self.model == other.model

return result

>>> my\_car = Car('Honda', 'Civic', 28)

>>> his\_car = Car('Honda', 'Civic', 28)

>>> my\_car.same\_car(his\_car)

True

However it would be nice if we did not have to create a new method name for such a method.

**Change the meaning of a operator**

It would be nice to use the **== and have it mean 'same make and model'** for our car objects.

And we can do that!

We just use the magic method **\_\_eq\_\_.** It **defines the behavior for the equality operator, ==.**

def \_\_eq\_\_(self, other):

result = self.make == other.make and self.model == other.model

return result

When we have a method that deals with two objects, we typically have two parameters: **self**

**and other.**

And now we can simply use == on cars:

>>> my\_car = Car('Honda', 'Civic', 28)

>>> his\_car = Car('Honda', 'Civic', 28)

>>> your\_car = Car('Porsche', '911', 25)

>>> his\_car == my\_car

True

>>> your\_car == my\_car

False

When we write: your\_car == my\_car

Python invokes:

your\_car. \_\_eq\_\_(my\_car)

When we write: his\_car == my\_car

Python invokes:

his\_car. \_\_eq\_\_(my\_car)

And here's where the magic is!

**11.4. Rich Comparison Methods**

At the beginning of this course, we introduced the following comparison operators supported by Python: ==, !=, <, >, <=, >=.

We have used these operators on the built in Python types integers and floats.

Python allows us to **define the behavior of all of these operators on our own objects** by defining the special methods below:

\_\_eq\_\_(self, other) == equality operator

\_\_ne\_\_(self, other) != inequality operator

\_\_lt\_\_(self, other) < less-than operator

\_\_gt\_\_(self, other) > greater-than operator

\_\_le\_\_(self, other) <= less-than-or-equal-to operator

\_\_ge\_\_(self, other) >= greater-than-or-equal-to operator

**When we write: Python invokes:**

x == y x.\_\_eq\_\_(y)

x != y x.\_\_ne\_\_(y)

x < y x.\_\_lt\_\_(y)

x > y x.\_\_gt\_\_(y)

x <= y x.\_\_le\_\_(y)

x >= y x.\_\_ge\_\_(y)

We have seen how to define \_\_eq\_\_ method for our Car class.

However x == y being True does not imply that x != y is False.

So when we define \_\_eq\_\_ we should also define \_\_ne\_\_ so that the operators will behave as expected.

Let’s go back and define a \_\_ne\_\_ method for our cars.

When we write x != y, Python will invoke: x.\_\_ne\_\_(y)

\_\_ne\_\_ compares two objects, so we need two parameters: self and other.

def \_\_ne\_\_(self, other):

result = self.make != other.make or self.model != other.model

return result

And we can test our new magic method as follows:

>>> my\_car = Car('Honda', 'Civic', 28)

>>> his\_car = Car('Honda', 'Civic', 28)

>>> your\_car = Car('Porsche', '911', 25)

>>> his\_car == my\_car

True

>>> his\_car != my\_car

False

>>> your\_car == my\_car

False

>>> your\_car != my\_car

True

Now let's define the \_\_lt\_\_ (less than) method on our cars based on their mileage.

Remember that when we have a method that deals with two objects, we have 2 parameters: self and other.

def \_\_lt\_\_(self, other):

return self.mileage < other.mileage

And we can test it as follows:

>>> from car import Car

>>> my\_car = Car('Honda', 'Civic', 28)

>>> my\_car.add\_gas(15)

<car.Car object at 0x00000000036B9E10>

>>> my\_car.drive(35)

<car.Car object at 0x00000000036B9E10>

>>> his\_car = Car('Porsche', '911', 25)

>>> his\_car.add\_gas(12)

<car.Car object at 0x00000000036B9E10>

>>> his\_car.drive(200)

<car.Car object at 0x00000000036CF668>

>>> my\_car < his\_car

True

>>> his\_car < my\_car

False

When I write:

my\_car < his\_car

Python invokes:

my\_car.\_\_lt\_\_(his\_car)

When I write:

his\_car < my\_car

Python invokes:

his\_car.\_\_lt\_\_(my\_car)

**< and > Conversions?**

If we define a \_\_lt\_\_ method but no \_\_gt\_\_ method, Python will use the \_\_lt\_\_ method with operands swapped. **Python会自动将 A>B 换为B<A, 如果\_gt\_内设方法没有对应\_lt\_的定义**

>>> his\_car > my\_car

True

Python converts it to: my\_car < his\_car and calls my\_car.\_\_lt\_\_(his\_car).

**However, Python will not combine methods.**

If we define a \_\_lt\_\_ method and an \_\_eq\_\_ method and try to test whether x <= y, Python **will NOT** call \_\_lt\_\_ and \_\_eq\_\_.

>>> my\_car <= his\_car

Traceback (most recent call last):

File "<input>", line 1, in <module>

TypeError: unorderable types: Car() <= Car()

The module functools provides a way to define all rich comparison methods if we only define \_\_eq\_\_ and one other method.

**11.5. Arithmetic & Logical Operators Methods**

Python also allows us to **define the behavior of arithmetic and logical operators** on our own objects by defining the special methods below:

**When we write: Python invokes:**

x + y x.\_\_add\_\_(y)

x - y x.\_\_sub\_\_(y)

x \* y x.\_\_mul\_\_(y)

x / y x.\_\_truediv\_\_(y)

x \*\* y x.\_\_pow\_\_(y)

x & y x.\_\_and\_\_(y)

x ^ y x.\_\_xor\_\_(y)

x | y x.\_\_or\_\_(y)

Since adding and subtracting cars does not make much sense, let's define an account class so that we can demonstrate the use of these magic methods:

class Account(object):

"""

Represent a bank account.

Argument:

account\_holder (string): account holder's name.

Attributes:

holder (string): account holder's name.

balance (float): account balance in dollars.

"""

def \_\_init\_\_(self, account\_holder):

self.holder = account\_holder

self.balance = 0

def \_\_str\_\_(self):

return self.holder + ': $' + str(self.balance)

def \_\_add\_\_ (self, other):

new\_holder = self.holder + ' & ' + other.holder

new\_account = Account(new\_holder)

new\_balance = self.balance + other.balance

new\_account.deposit(new\_balance)

return new\_account

def deposit(self, amount):

"""

**deposit the given amount to the account**

Parameter:

amount (float): the amount to be deposited in dollars.

**Returns:**

**the updated account object**

"""

**self.balance += amount**

**return self**

We defined our account addition such that a new account object is created. The new account object is a joint account of the two individual holders and has a balance equal to the sum of the balances of the individual accounts.

We also defined a \_\_str\_\_ method so we can print our account objects

Note here that the \_\_add\_\_ method is creating a new account object are returning it. It is NOT altering self or other.

When we write:

their\_account = her\_account + his\_account

Python will invoke:

their\_account = her\_account.\_\_add\_\_(his\_account)

Now we can test our addition of account objects:

>>> from account import Account

>>> my\_account = Account('Rula')

>>> his\_account = Account('Bob')

>>> my\_account.deposit(20)

<account.Account object at 0x000000000328CDD8>

>>> his\_account.deposit(100)

<account.Account object at 0x000000000328C9B0>

>>> print (my\_account)

>>> print(my\_account)

Rula: $20

>>> print(his\_account)

Bob: $100

>>> print(my\_account + his\_account)

>>> print(my\_account + his\_account)

Rula & Bob: $120

**Python turned my\_account + his\_account into:**

**my\_account.\_\_add\_\_(his\_account)**

**operator.\_\_add\_\_(*a*, *b*)**

**Return a + b, for *a* and *b* numbers**

**11.6. Indexing**

The **\_\_getitem\_\_ ()** **and \_\_setitem\_\_()** methods allow us ***to emulate indexing*** on our object.

Let's see how \_\_getitem\_\_() works with an example:

We'll start by creating a Book class.

class Book(object):

"""

Represent a book

Arguments:

author (string): the author's name

title (string): the book title

Attributes:

author (string): the author's name

title (string): the book title]

content (list): list containing the content of each chapter

"""

def \_\_init\_\_(self, author, title):

self.author = author

self.title = title

self.content = []

def \_\_str\_\_(self):

result = self.title + ' by: ' + self.author

chapter\_number = 1

# add chapter numbers to the representation

for chapter in self.content:

result += '\nChapter ' + str(chapter\_number) + '\n' + chapter

chapter\_number += 1

return result

def \_\_getitem\_\_(self, key):

# if the index is in the existing chapters range

if 0 < key <= len(self.content):

return self.content[key - 1] # **convert to 0 based indexing , 这里 如果我们查看Chapter 1内容, 只需输入book.\_getitem\_(self, 1), 返回 self.content[1-1] 即 self.content[0]**

def add\_chapter(self, text):

self.content.append(text)

A book object has an author and a title but it is also a container for the chapters. The chapters are kept in a list. We would like to be able to access chapter 1 of a book by writing my\_book[1] instead of my\_book.content[0].

Defining **\_\_getitem\_\_** as shown above allows us to do that. Now we can test our class as follows:

>>> from book import Book

We first create a book object.

>>> my\_book = Book('Rula Khayrallah', 'Python is fun!')

Let's print it to see that everything is there.

>>> print(my\_book)

Python is fun! by: Rula Khayrallah

Now let's add three chapters.

>>> my\_book.add\_chapter('Introducing Python and Our Environment')

>>> my\_book.add\_chapter('Python Basics')

>>> my\_book.add\_chapter('Lists')

Let's print to check that everything is there.

>>> print(my\_book)

Python is fun! by: Rula Khayrallah

Chapter 1

Introducing Python and Our Environment

Chapter 2

Python Basics

Chapter 3

Lists

Now to access chapter 2, we can write:

>>> print(my\_book[2])

Python Basics

To access chapter 1, we can write:

>>> print(my\_book[1])

Introducing Python and Our Environment

And chapter 3:

>>> print(my\_book[3])

**Lists**

Note that **\_\_geitem\_\_() implements read access only**. If we want to be able to update chapter 1 by writing

my\_book[1] = ....

we would have to implement a \_\_setitem\_\_() method in our class.

**12.1. Why Inheritance?**

When we approach a problem from an object oriented perspective, we often find that different abstract data types are related. Similar classes may differ in their amount of specialization.

**12.2. A Bank Account Example**

To illustrate the concept of inheritance, let’s go back to our bank account object that we briefly introduced in section 11.5.

#-------------------------------------------------------------------------------

# Name: account

# Purpose: contains the class definitions for bank accounts

#

# Author: Rula Khayrallah

#

# Created: 10/04/2013

#-------------------------------------------------------------------------------

"""

Module to describe and manipulate bank accounts.

"""

class Account(object):

"""

Represent a bank account.

Argument:

account\_holder (string): account holder's name.

Attributes:

holder (string): account holder's name.

balance (float): account balance in dollars.

"""

def \_\_init\_\_(self, account\_holder):

self.holder = account\_holder

self.balance = 0

def \_\_str\_\_(self):

return self.holder + ': $' + str(self.balance)

def deposit(self, amount):

"""

deposit amount to the account

Parameter:

amount (float): the amount to be deposited in dollars.

Returns:

the updated account object

"""

self.balance += amount

return self

def withdraw(self, amount):

"""

withdraw the amount from the account if possible.

Parameter:

amount (float): the amount to be withdrawn in dollars.

Returns:

boolean: True if the withdrawal is successful

False otherwise

"""

if self.balance >= amount:

self.balance = self.balance - amount

return True

else:

return False

Having defined the class Account with the methods deposit and withdraw, we can now use them as follows:

>>> from account import Account

>>> my\_account = Account('Rula')

>>> my\_account.deposit(20)

<account.Account object at 0x0000000003EF0128>

>>> my\_account.withdraw(10)

True

>>> print(my\_account)

Rula: $10

>>> my\_account.withdraw(15)

False

Now let’s assume that the bank introduces a new type of account: a savings account which charges us an extra fee for withdrawal.

The bank just wants to encourage us to save our money and not withdraw it.

This savings account behaves the same way as the general bank account we saw earlier for deposits.

Withdrawals are a little different since that special fee has to be included in the computation.

Ideally, we would like the following behavior:

>>> her\_account = SavingsAccount('Julie')

The deposit behavior should be the same as that of the regular account.

>>> her\_account.deposit(20)

<account.SavingsAccount object at 0x0000000003EFA390>

Withdrawals are different. We'll assume a withdrawal fee of $1. When we withdraw $10, $11 are actually taken out of the account.

>>> her\_account.withdraw(10)

True

>>> print(her\_account)

Julie: $9

>>> her\_account.withdraw(15)

False

To implement the SavingsAccount class, it would be nice if we could use the same method names deposit and withdraw.

We would like deposit to be the same as that for the generic account.

We don’t want to repeat the code for deposit – remember DRY?

We would like withdraw to be different without changing the behavior of the generic account.

We can do that with inheritance.

We write our SavingsAccount class definition as follows:

class SavingsAccount(Account):

"""

Represent a savings bank account with a withdrawal fee.

"""

This class definition indicates that the new class is a SavingsAccount, which is a kind of Account.

This 'kind of…class' appears between the parentheses.

Remember we had object there before…

The class Account (generic account) is now the **base class** of SavingsAccount.

The base class is also called parent class and superclass.

The class SavingsAccount is a subclass of Account.

The subclass is also called child class.

**Inheritance is used to indicate that one class will get most or all of its features from a parent class.**

**This happens implicitly whenever we write: class SavingsAccount(Account):**

It means : make a class SavingsAccount that inherits from Account.

More specifically, we want SavingsAccount to inherit the deposit method from Account.

This is done by default. We don’t have to do anything in the SavingsAccount class.

We can now simply invoke deposit on a SavingsAccount instance:

>>> from account import SavingsAccount

>>> her\_account = SavingsAccount('Julie')

>>> her\_account.deposit(20)

<account.SavingsAccount object at 0x0000000003EFA390>

>>> print(her\_account)

Julie: $20

Note that the \_\_str\_\_ method is also inherited by the SavingsAccount class.

We'll encounter many classes with similar attributes, where one class represents a special case of the other. Inheritance allows us to specify only what is different between the more specialized class and the more general one.

**12.3. Override or Inherit?**

A subclass inherits the attributes of its base class, but may override certain attributes, including certain methods.

We would like to override the withdraw method and implement our fee based withdrawal for SavingAccount. To achieve that we do the following:

We introduce a class variable, fee, that is specific to the SavingsAccount class. The fee class variable does not exist in the Account class.

We also define a new withdraw() method to override the behavior defined in the Account class.

All other behavior is inherited from the base class Account. That includes \_\_init\_\_ , deposit and \_\_str\_\_.

class SavingsAccount(Account):

"""

Represent a savings bank account with a withdrawal fee

Argument:

account\_holder (str): account holder's name.

Attributes:

holder (str): account holder's name.

balance (float): account balance in dollars.

"""

fee = 1

def withdraw(self, amount):

"""

withdraw the amount and the fee from the account if possible.

Parameter:

amount (float): the amount to be withdrawn in dollars.

Returns:

boolean: True if the withdrawal is successful

False otherwise

"""

if self.balance >= amount + self.fee:

self.balance = self.balance - amount - self.fee

return True

else:

return False

**With inheritance, we only specify what is different between the child and the parent class.**

Anything that we leave unspecified in the child class is automatically assumed to behave just as it would for the parent class.

Now we can use the SavingsAccount class as follows:

>>> her\_account = SavingsAccount('Julie')

>>> her\_account.deposit(20)

<account.SavingsAccount object at 0x0000000003EFA390>

>>> her\_account.withdraw(10)

True

>>> print(her\_account)

Julie: $9

>> her\_account.withdraw(15)

False

**12.4. IS-A Relationship**

It's important to note that **inheritance is between classes, not between objects.**

A parent class is a template that is used to create objects.

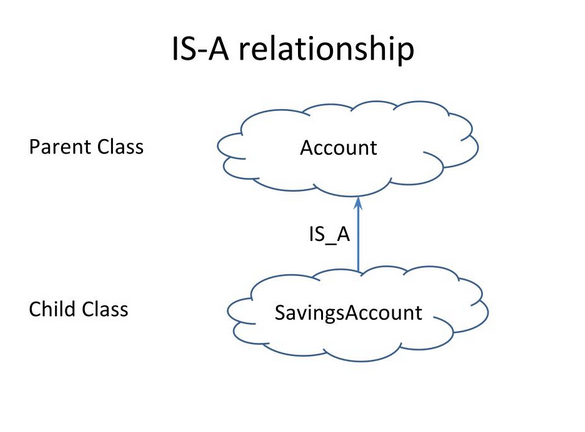
A child class of the parent is another template that looks a lot like the original, but with some added features.

The child class is used to create objects that look like the parent's objects, but with added features.

**Inheritance represents an is-a relationship between classes.**

Our savings account class inherits from our account class.

A savings account **is-a specific kind of account**.



**12.5. Override and Inherit?**

Sometimes there is a special case where we first override a method to define some specialized behavior of the child class but then we also want to invoke the parent version on the child instance.

Suppose that we want to introduce a PremiumAccount that is an interest bearing account. The bank will pay us interest on our money.

**PremiumAccount is still a kind of generic account but we would want to add an attribute, interest\_rate just for this kind of account.**

We don’t want to change our base class Account.

In this case we can define an \_\_init\_\_ () method for the subclass PremiumAccount as follows:

class PremiumAccount(Account):

"""

Represent a premium interest bearing bank account.

Argument:

account\_holder (str): account holder's name.

rate (float): interest rate

Attributes:

holder (str): account holder's name.

balance (float): account balance in dollars.

interest\_rate (float): interest rate

"""

def \_\_init\_\_(self, account\_holder, rate):

self.interest\_rate = rate

self.holder = account\_holder

self.balance = 0

Note that the **last two lines are identical to the lines in the Account class \_\_init\_\_**.

Remember our **DRY principle: Don’t Repeat Yourself?**

What if we could call the Account class \_\_init\_\_ to perform the rest of the initialization?

There are two ways to do that:

* **using super()**
* **through the class name**

We can define an \_\_init\_\_ method for the subclass PremiumAccount as follows:

**def \_\_init\_\_(self, account\_holder, rate):**

**self.interest\_rate = rate**

**super().\_\_init\_\_(account\_holder)**

Here we first add the interest\_rate initialization in the child class, then we invoke the \_\_init\_\_ method in the parent class to complete the initialization.

super() gets the **super class (parent class) of the current class**.

>>> from account import PremiumAccount

>>> my\_account = PremiumAccount('Rula', 0.01)

>>> print(my\_account)

Rula: $0

>>> my\_account.interest\_rate

0.01

>>> my\_account.deposit(20)

<account.PremiumAccount object at 0x000000000360CF28>

>>> print(my\_account)

Rula: $20

**Another way invoke the parent \_\_init\_\_ method is through the class name:**

def \_\_init\_\_(self, account\_holder, rate):

self.interest\_rate = rate

**Account.\_\_init\_\_(self, account\_holder)**

Note that when we invoke a parent method through the class name, we have to add self as an argument.

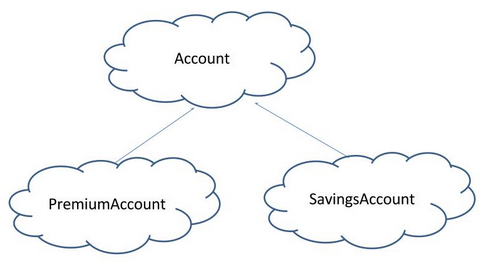
The most common use of super() is actually in \_\_init\_\_ methods. This is usually the only place where we need to do some things in a child, then complete the initialization in the parent.

However super() may be used in any method to access a parent method.

**12.6. Multiple Inheritance**

We have **multiple inheritance when a class inherits from more than one class.** Python supports multiple inheritance.

Here are the classes we have created so far:



Now suppose we want to create a new type of account, the Premium Savings Account.

This account is an interest bearing account but it charges us a fee every time we make a withdrawal.

This account combines features from the Savings Account and from the Premium Account.

Now let’s define the new class PremiumSavingsAccount:

class PremiumSavingsAccount(PremiumAccount, SavingsAccount):

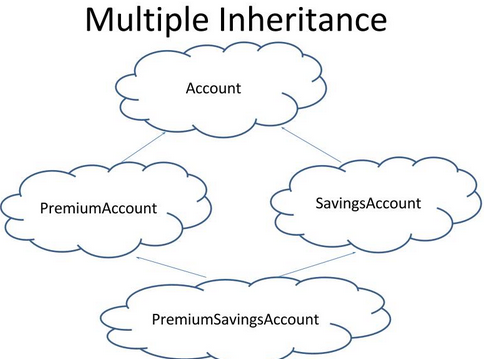
"""

Represent a premium interest bearing bank account with a withdrawal fee

"""

… and that’s all we need to do.

**The PremiumSavingsAccount will inherit the right behavior from the parent classes PremiumAccount and SavingsAccount as well as from Account.**



Let’s test it out:

>>> his\_account = PremiumSavingsAccount('Bob', 0.015)

>>> print(his\_account)

Bob: $0

>>> his\_account.interest\_rate

0.015

>>> his\_account.deposit(50)

<account.PremiumSavingsAccount object at 0x00000000033E5358>

>>> print(his\_account)

Bob: $50

>>> his\_account.withdraw(10)

True

>>> print(his\_account)

Bob: $39

The interest\_rate is initialized in the PremiumAccount class.

The \_\_str\_\_ method is defined in the Account class.

The deposit method is defined in the Account class.

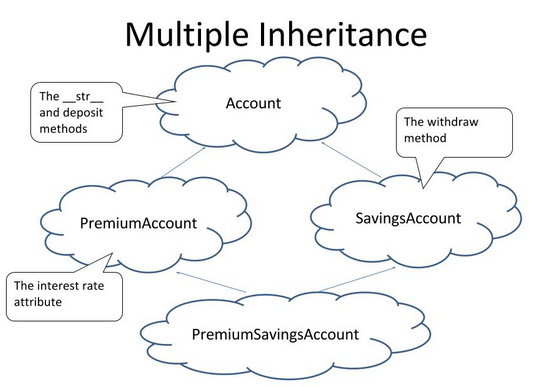
The withdraw method is defined in the SavingsAccount class.

Python uses an algorithm called the **C3 method resolution order (MRO)** to figure out multiple inheritance.

In this particular example (**diamond shaped inheritance), Python resolves names from left to right, then upwards.**

So Python checks for an attribute/method name in the following classes, in order, until an attribute with that name is found:

**PremiumSavingAccount → PremiumAccount, →SavingAccount→Account→Object.**



**12.7. isinstance and issubclass**

The **built-in function** **isinstance** may be used to **check if an object is an instance of a certain class** or **of a direct or indirect subclass of that class**.

**It is very useful when the inheritance chain is not that obvious.**

Let's try it on our various account objects.

**We'll use the \* to import all the various class definitions contained in the account module.**

>>> **from account import \***

Let's start with a savings account:

>>> a = SavingsAccount('Alice')

>>> isinstance(a, SavingsAccount)

True

>>> isinstance(a, Account)

True

>>> isinstance(a, PremiumAccount)

False

>>> isinstance(a, PremiumSavingsAccount)

False

>>> isinstance(a, object)

True

Now let's try a premium savings account:

>>> b = PremiumSavingsAccount('Bob', 0.05)

>>> isinstance(b, PremiumSavingsAccount)

True

>>> isinstance(b, SavingsAccount)

True

>>> isinstance(b, PremiumAccount)

True

>>> isinstance(b, Account)

True

>>> isinstance(b, object)

True

We can also use isinstance on the built-in types:

>>> isinstance(500, int)

True

>>> isinstance(500, float)

False

>>> isinstance(500, bool)

False

>>> isinstance(500, str)

False

>>> isinstance(500, dict)

False

>>> isinstance(500, list)

False

>>> isinstance(500, object)

True

>>> isinstance(True, int)

True

>>> isinstance(True, bool)

True

issubclass is another built-in function that works with inheritance.

It returns True if a certain class is a subclass – direct or indirect – of another class

>>> issubclass(PremiumAccount, Account)

True

>>> issubclass(PremiumAccount, SavingsAccount)

False

>>> issubclass(PremiumSavingsAccount, Account)

True

>>> issubclass(SavingsAccount, SavingsAccount)

True

It also works with the built-in types:

>>> issubclass (int, float)

False

>>> issubclass (float, int)

False

**>>> issubclass (bool, int)**

True

>>> issubclass (int, bool)

False

**12.8. Composition**

Object composition is a way to construct objects out of other components.

Objects in one class contain references to objects in another class.

Suppose I have an Employee class that I use for employees. In this class, each instance has a name, a job description and a bank account.

#-------------------------------------------------------------------------------

# Name: employee

# Purpose: contains the class definitions for the Employee class

#

# Author: Rula Khayrallah

#

# Created: 10/3/2014

#-------------------------------------------------------------------------------

"""

Module to describe employees.

"""

**from account import Account**

class Employee(object):

"""

Represent an employee.

Argument:

name (string): employee's name.

Attributes:

name (string): employee's name.

job (string): employee's job description

account (Account): employee's bank account

"""

def \_\_init\_\_(self, name):

self.name = name

self.job = ''

**self.account = Account(name) // Account class**

This kind of relationship is called a **HAS-A relationship**.

**A person has an account.**

It’s just another way of **using code written for one class in another class.**

>>> from employee import Employee

>>> new\_hire = Employee('Alex')

>>> new\_hire.account.**deposit(500)**

<account.Account object at 0x00000000033DD748>

>>> print(new\_hire.account)

Alex: $500

**13. Advanced OOP Topics**

**13.1. Encapsulation and Attribute Access 封装 和 属性access**

We've seen that Python **does NOT** enforce encapsulation since **we have access to all the attributes of a given object from outside the class.**

However **Python provides some special methods, that if defined, control this access**:

**\_\_getattribute\_\_** object.\_\_getattribute\_\_(*self*, *name*)

**\_\_setattribute\_\_** object.\_\_setattr\_\_(*self*, *name*, *value*)

**\_\_getattr\_\_** object.\_\_getattr\_\_(*self*, *name*)

**The \_\_getattribute\_\_ method, if defined, is called whenever we access an attribute.**

So if \_\_getattribute\_\_ is defined, whenever we write:

**x.y**

Python will invoke

**x.\_\_getattribute\_\_(y)**

Python will call \_\_getattribute\_\_ every time an attribute or method name is referenced on an object (**except magic method names, since that would cause an infinite loop**).

**The \_\_setattr\_\_ method, if defined, is called whenever we assign a value to an attribute.**

It is called **regardless of whether or not that attribute exists**.

So whenever we write:

**x.y = value**

Python will invoke

**x.\_\_setattr\_\_('y', value)**

We have to be very careful with \_\_getattribute\_\_ and \_\_setattr\_\_ because **these methods are called every time an attribute is accessed or assigned, even from within these methods**:

Consider the following example:

def \_\_setattr\_\_ (self , **name , value** ):

**self.name = value**

Here the method keeps calling itself (because we are setting the name attribute) and the recursion goes on forever… 无限循环

**To avoid the infinite recursion, we can call the parent’s class \_\_setattr\_\_() method**.

Here's an example where we use the **\_\_setattr\_\_ method to limit access to the balance attribute of an Account object**:

def \_\_setattr\_\_(self, key, value):

if key == 'balance':

print('This is a read-only attribute')

else:

**super().\_\_setattr\_\_(key, value)** ***// 如果super class里的attribute 值未被赋予, then 对super class 使用 一次 setattr方法来赋值, 想等后 再次赋值会被提示'This is a read-only attribute'***

Note that now we also need to change all the existing methods that set the balance, including \_\_init\_\_, deposit and withdraw. These methods will also need to use the the parent’s class \_\_setattr\_\_() method because self.balance = 0 will not work.

def \_\_init\_\_(self, account\_holder):

self.holder = account\_holder

**super().\_\_setattr\_\_('balance', 0)**

def deposit(self, amount):

"""

deposit amount to the account

Parameter:

amount (float): the amount to be deposited in dollars.

Returns:

the updated account object

"""

**super().\_\_setattr\_\_('balance', self.balance + amount)**

return self

And now we can test the balance access as follows:

>>> from account import Account

>>> my\_account = Account('Rula')

I can read the balance:

>>> my\_account.balance

0

I cannot set it.

>>> my\_account.balance = 20

This is a read-only attribute

**Other attributes' access is not affected.**

>>> my\_account.holder = 'Bob'

>>> print(my\_account)

Bob: $0

The special method **\_\_getattr\_\_** may be **used to define** behavior for when a user attempts to **access an attribute that doesn't exist**.

This can be useful for catching and redirecting common misspellings, giving warnings about using old attributes. **It only gets called when a nonexistent attribute is accessed.**

**13.2. Properties**

A decorator in Python allows us to modify the behavior of the method, function or class that follows it. We can define our own decorators (this is beyond the scope of this course) or use some of the predefined decorators. **@property is one of these predefined decorators**.

The decorator **@property**, allows methods to be called without the standard call expression syntax. This provides **a simple way for computing attributes on the fly**.

Let's make sense of this with an example.

We'll add a method to our Car class that returns the maximum distance we can drive with the current gas in the tank:

def how\_far\_can\_we\_go(self):

""" return maximum distance we can drive"""

return self.fuel\_efficiency \* self.gas\_in\_tank

This is a method, so here’s how we would invoke it:

>>> my\_car = Car('Honda', 'Civic', 28, 10000, 5)

>>> my\_car.how\_far\_can\_we\_go()

140

Sometimes it is useful to be able to access how\_far\_we\_can\_go like an attribute such as mileage or make or model.

Since it may be computed on the fly from the current instance variables fuel\_efficiency and gas\_in\_tank, we don’t really need to add it as another attribute and store it in the object.

So we just define it as a property by adding the decorator **@property** right before its definition.

**@property**

**def how\_far\_can\_we\_go(self):**

**""" return maximum distance we can drive"""**

**return self.fuel\_efficiency \* self.gas\_in\_tank**

And now we can write:

>>> my\_car = Car('Honda', 'Civic', 28, 10000, 5)

>>> my\_car.how\_far\_can\_we\_go

140

We don’t need the parentheses here.

Note however that we **cannot assign a value** to it.

>>> my\_car.how\_far\_can\_we\_go = 0

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

AttributeError: can't set attribute

**Properties are sometimes also used to create pseudo read-only attributes.**

Going back to our Account class, another way to limit access to the balance attribute is to use **\_balance**. The **underscore** indicates that this is a **'pseudo' private attribute**.

Then we would define a balance property so that **users can read the balance but not modify it**. Note that **\_balance** is **NOT** included in the docstring.

class Account(object):

"""

Represent a bank account.

Argument:

account\_holder (string): account holder's name.

Attributes:

holder (string): account holder's name.

"""

def \_\_init\_\_(self, account\_holder):

self.holder = account\_holder

self.\_balance = 0 **# 私有属性/私有变量**

def deposit(self, amount):

"""

deposit amount to the account

Parameter:

amount (float): the amount to be deposited in dollars.

Returns:

the updated account object

"""

self.\_balance += amount

return self

**@property**

**def balance(self):**

**return self.\_balance**

**Defining a balance property lets users read the balance as an attribute but not modify it.**

>>> from account import Account

>>> alice = Account('Alice')

>>> her\_account = Account('Alice')

>>> her\_account.deposit(50)

<account.Account object at 0x0000000003E9C898>

>>> her\_account.balance

50

>>> her\_account.balance = 100

Traceback (most recent call last):

File "<input>", line 1, in <module>

AttributeError: can't set attribute

**13.3. Static Methods**

All of the methods we have seen so far are **instance methods**. **They act on one instance**. That instance is accessible to the method via the **parameter** **self.** The instance methods access the instance variables and may or may not mutate them.

There are cases where **methods may not need to access or mutate any of the information stored in the instance**. **However we still want to be able to invoke these methods on instances of the given class**. **We also want these methods to be inherited by subclasses of the given class. 静态方法 不access或改变instance内部信息, 但是我需要对指定的类使用此方法, 并且 衍生类也需要承接此方法**

We can write these methods as **static methods**.

Let's illustrate this with a new class to represent students.

Students have a name and a student id. We need to validate the student id before storing it in the object so we write a static method that takes an id as a parameter and returns True if that id is valid and False otherwise, You can see that this static method is more like a function in that it does not really need the object.

#-------------------------------------------------------------------------------

# Name: student

# Purpose: contains the class definitions for students

#

# Author: Rula Khayrallah

#

# Created: 10/12/2014

#-------------------------------------------------------------------------------

"""

Module to describe students in a college setting.

"""

class Student(object):

"""

Represent a student.

Arguments:

name (string): student name

sid (int): student id - 8 digits

Attributes:

name (string): student name

sid (int): student id - 8 digits

"""

def \_\_init\_\_(self, name, sid):

self.name = name

if self.valid(sid):

self.sid = sid

else:

self.sid = 0

@staticmethod

def valid(some\_id):

"""" A valid student id starts with 2014 """ # **这个方法只做判定T/F, 没有改变对象中的信息**

if some\_id // 10000 == 2014:

return True

else:

return False

To tell the interpreter that a given method is a static method (and **hence does not need to be passed the parameter self**), we include the decorator **@staticmethod** before the method definition. @staticmethod is a predefined decorator, just like @property.

**静态方法，既不需要self（对象实例object instance）也不需要cls（类）作为隐藏参数被传递**。

Now we can test our class definition as follows:

>>> from student import Student

>>> john = Student('John', 20141111)

>>> lynn = Student('Lynn', 20909909)

>>> john.sid

20141111

>>> lynn.sid

0

Since Lynn's id was invalid, it was replaced by 0.

**3.4. Class Methods**

There are also cases where **methods may not need to access or mutate any of the information stored in the instance** **but they need to** **access some information specific to the class**. We could write these methods **as class methods**. 不改变或涉及instance里的信息, 但是要access到 类中的信息, **称为类方法**

Let's illustrate this with our Student class.

Let's assume that we need to keep track of the total number of students.

We'll add **a class variable, number\_of\_students that we'll update whenever a new instance is initialized. 类变量**

We'll write a method to update it. That method does not need the instance. It only needs the class. We can write it as follows:

#-------------------------------------------------------------------------------

# Name:        student

# Purpose:     contains the class definitions for students

#

# Author:      Rula Khayrallah

#

# Created:     10/12/2014

#-------------------------------------------------------------------------------

"""

Module to describe students in a college setting.

"""

class Student(object):

    """

    Represent a student.

    Arguments:

    name (string): student name

    sid (int): student id - 8 digits

    Attributes:

    name (string): student name

    sid (int): student id - 8 digits

    """

number\_of\_students = 0 # 类变量

    def \_\_init\_\_(self,  name, sid):

        self.name = name

        if **self.valid(sid):**

            self.sid = sid

        else:

            self.sid = 0

**self.update\_count()**

    @staticmethod

    def valid(some\_id):

        """" A valid student id starts with 2014 """

        if some\_id // 10000 == 2014:

            return True

        else:

            return False

**@classmethod**

    def update\_count(**cls**):

        cls.number\_of\_students += 1

To tell the interpreter that a given method is a class method (**and hence needs the to be passed the class of the instance and not the instance itself**) **#类的对象实例作为隐式的参数传递给第一个参数而不是self**

we include the decorator @classmethod before the method definition. @classmethod is another predefined decorator.

Now we can test our class definition as follows:

>>> from student import Student

>>> alice = Student('Alice', 20143333)

>>> bob = Student('Bob', 20144444)

>>> Student.number\_of\_students

2

**14.1. What is an Exception?**

An exception is an indication that something went wrong.

Some programming languages encourage the use of error return codes, which you check. Python encourages the use of exceptions, which you handle.

So far, working with Python, we have encountered many errors. Not all errors are exceptions.

Syntax errors are the most common kind of errors you get when you are learning Python.

Let's say I forget the colon in an if statement:

if i > 5

print(i)

File "....", line 1

if i > 5

^

SyntaxError: invalid syntax

The problem location is highlighted in the message.

We can also get syntax errors in the interpreter:

>>> print 5

File "<input>", line 1

print 5

^

SyntaxError: invalid syntax

**Syntax errors are detected even if they are in a portion of the program that is never executed.**

def never\_called():

print 'hi'

def main():

print('hello')

if \_\_name\_\_ == '\_\_main\_\_':

main()

File "...", line 2

print 'hi'

^

SyntaxError: invalid syntax

PyCharm or any other good IDE (or test editor) help us avoid syntax errors by detecting them and highlighting them as we type in our code.

However even if the syntax of a statement or expression is correct, it may cause an error when we try to execute it. **Errors detected during execution are called exceptions and are not necessarily fatal**: we will learn how to **handle** them in Python programs.

Here are a few examples:

**An attempt to divide by 0 results in an exception.**

>>> total = 100

>>> number = 0

>>> print (total / number)

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

ZeroDivisionError: division by zero

**The last line of the error message indicates what happened.**

**The previous lines indicate where it happened (traceback)**

Exceptions come in different types (or classes), and the **type is printed as part of the message.**

The exception type above is **ZeroDivisionError**. It is the name of the built-in exception that occurred.

**An attempt to use a variable that has not been assigned a value results in an exception.**

>>> print (novariable)

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

**NameError**: name 'novariable' is not defined

**The exception type above is NameError**.

**An attempt to add a string to an integer results in an exception**.

>>> total = '5' + 3

**Traceback (most recent call last):**

**File "<string>", line 301, in runcode**

**File "<interactive input>", line 1, in <module>**

**TypeError: Can't convert 'int' object to str implicitly**

Here the exception type is **TypeError**.

An attempt to open a file that does not exist results in an exception.

with open('nofile') as file:

file.read()

Traceback (most recent call last):

File "<string>", line 301, in runcode

File "<interactive input>", line 1, in <module>

FileNotFoundError: [Errno 2] No such file or directory: 'nofile‘

The exception typeabove is **FileNotFoundError**.

Note that unlike syntax errors, **an exception is only raised** if the corresponding statement is **executed**.

The code below does not result in an exception:

def never\_called():

print(5/0)

def main():

print('hello')

if \_\_name\_\_ == '\_\_main\_\_':

main()

hello

There are many more built-in exceptions. You can look them up when needed. Here are just a few more:

IndexError

KeyError

FileExistsError

KeyboardInterrupt

**14.2. Handling Exceptions**

An exception doesn’t need to result in a program crash. It can be handled.

Sometimes an exception is due to a bug in our code (like accessing a variable that does not exist), but sometimes an exception is something we can anticipate.

If we are opening a file, that file might not exist.

If we are prompting the user for input, that input may be invalid.

If we know that a line of code may raise an exception, we must **handle it** using a **try...except** block.

Consider the following:

def main():

grade = float(input('Please enter your grade: '))

if \_\_name\_\_ == '\_\_main\_\_':

main()

What happens if the user accidentally enters a string?

Please enter your grade: hello

Traceback (most recent call last):

File "...", line 8, in <module>

main()

File "...", line 5, in main

grade = float(input('Please enter your grade: '))

ValueError: could not convert string to float: 'hello'

This is how we handle the exception:

def main():

**try:**

grade = float(input('Please enter your grade: '))

**except** ValueError:

print("That's not a valid grade. Please try again.")

if \_\_name\_\_ == '\_\_main\_\_':

main()

Note the colons after the try and the except keywords. Note also the indentation of the subsequent blocks. The try and except clauses are at the same indentation level.

The interpreter **starts by executing the try** clause.

**If all goes well, it skips the except clause** and proceeds.

**If an exception occurs, it jumps out of the try clause**.

**Then if the exception type matches the exception** named after the except keyword**, the except clause is executed**.

**If an exception occurs which does not match the exception named in the except clause, and no handler is found,** it is an **unhandled exception** and **execution stops with a message**.

How do I know it is a ValueError exception that I need to handle? It was the exception type that was generated before we added the try except clause.

And here's our output now when we enter a string.

Please enter your grade: hello

That's not a valid grade. Please try again.

**14.3. The Exception Object**

We can catch the exception object by specifying **'as'.** The exception object (or instance) may contain more specific information about the exception. Here's an example:

def main():

try:

grade = float(input('Please enter your grade: '))

except ValueError as error:

print(str(error))

if \_\_name\_\_ == '\_\_main\_\_':

main()

Here error is the exception object.

Please enter your grade: hello

could not convert string to float: 'hello'

**14.4. Multiple except Clauses**

Sometimes one action may result in more than one type of errors.

Suppose that I am reading a list of numbers from a text file.

Many things may go wrong:

* The file might not exist.
* The file might not contain the expected data – in this case integers.

We can handle both of these issues with one try and multiple except clauses. Each one of these clauses corresponds to a different exception type.

try:

with open('myfile.txt', encoding='utf-8') as input\_file:

for line in input\_file:

i = int(line.strip())

except **IOError:**

print("I/O error")

except **ValueError:**

print("Could not convert data to an integer")

Here **at most one handler will be executed.**

Note that handlers only handle exceptions that occur in the corresponding try clause, not in other except clauses of the same try statement.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way!

try:

with open('myfile.txt', encoding='utf-8') as input\_file:

for line in input\_file:

i = int(line.strip())

except IOError as error:

print("I/O error:" + str(error))

except ValueError:

print("Could not convert data to an integer.")

**except:**  # all other cases

print('Unexpected error')

An except clause may also name multiple exceptions as **a parenthesized tuple**.

**except (RuntimeError, TypeError, NameError):**

**14.5. The else Clause**

The try … except statement has an optional else clause, which, when present, must follow all except clauses. It is useful **for code that must be executed if the try clause does not raise an exception**.

Using an else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the statement.

Here's an example:

more\_input = True

while more\_input:

try:

grade = float(input('Please enter your grade: '))

except ValueError:

print("That's not a valid grade. Please try again.")

else:

more\_input = False # 条件符合 , exit while loop

**14.6. The finally Clause**

The try statement has another optional clause, finally, which is intended to define **clean-up actions that must be executed under all circumstances. Finally 后面的必须执行, 任何情况下**

Consider the following function:

def divide(x, y):

try:

result = x / y

except ZeroDivisionError:

print("division by zero!")

else:

print("result is", result)

**finally:**

**print("finally... cleaning up now")**

Let's call the divide functions with valid and invalid arguments:

divide(10, 2)

result is 5.0

finally... cleaning up now

Here the finally block is executed when the try is successful and there is no exception.

divide(10, 0)

division by zero!

finally... cleaning up now

And here the finally block is executed when there is an exception and it is handled in the except clause.

divide(10, 'hello')

finally... cleaning up now

Traceback (most recent call last):

File "...", line 8, in divide

result = x / y

TypeError: unsupported operand type(s) for /: 'int' and 'str'

And here the finally block is executed when there is an unhandled exception.

As you can see, **the finally clause is executed in any event.**

The finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

|  |
| --- |
| **14.7. Raising Exceptions** |
|  |
|  |
| The raise statement allows us to **force a specified exception to occur.**  This may be an **exception class** such as NameError, ValueError or TypeError:  >>> raise NameError  Traceback (most recent call last):    File "<input>", line 1, in <module>  NameError  Or a particular **exception instance**.  **This allows us to provide a more specific message.**  >>> raise NameError ('The specified item does not exist')  Traceback (most recent call last):    File "<input>", line 1, in <module>  NameError: The specified item does not exist  raise (with no expression following it) is used to re-raise the active exception in an except clause.   try:          grade = float(input('Please enter your grade: '))  except ValueError:          print("That's not a valid grade.  Please try again.")  **raise**  Here we get the message followed by the traceback.  That's not a valid grade.  Please try again.  Traceback (most recent call last):    File "...."...  ValueError: could not convert string to float: 'hello' |

|  |
| --- |
| **14.8. User Defined Exceptions** |
|  |
|  |
| We can create our own exception class.  Exceptions must be a subclass of the **Exception** class, either directly or indirectly.  class NewError(**Exception**):      pass  Once we create a new exception class, we can raise it as follows:  raise NewError  Traceback (most recent call last):      raise NewError  \_\_main\_\_.NewError    or, if we want to specify a description:  raise NewError(**'oops!'**)  Traceback (most recent call last):      raise NewError('oops!')  \_\_main\_\_.NewError: oops!    And we can catch it in an except clause as follows:      try:          raise NewError('oops')      except **NewError as error**:          print (str(**error**))  oops  We can also **override the parent \_\_init\_\_  in the class definition** and provide some specific details as follows: 覆盖exception class的\_init\_方法定义  class NewError(Exception):        def \_\_init\_\_(self, code, message, severity):          self.code = code          self.message = message          self.severity = severity  Now to raise the exception with a code of 5, a message of 'Invalid request' and severity of 9, we write:  raise NewError (5, 'Invalid request', 9)  To catch it  we write:      try:          raise NewError (5, 'Invalid request', 9)      except NewError as **error**:          print (**error.message**, 'severity: ', **error.severity**)  Invalid request severity:  9  Because NewError is a subclass of Exception, specifying Exception instead of NewError in the except clause will also work:      try:          raise NewError (5, 'Invalid request', 9)      except **Exception** as error:          print (error.message, 'severity: ', error.severity)  Invalid request severity:  9 |

**15. Iterators & Generators**

**15.1. Sequences vs Iterators**

We have talked about sequence built-in data types such as tuples and lists.

Sequences support two operations: **length and indexing**.

Representing sequential data using the sequence abstraction has two limitations:

A sequence of length n usually takes up an amount of memory proportional to n. **So the longer a sequence is, the more memory it takes to represent it.**

Sequences can only represent datasets of known, **finite length.**

Iterators provide a construct for working with sequential data that can accommodate collections of unknown or **infinite length**, while using **limited memory**.

Our goal here is to represent sequential data without storing each element explicitly in the memory. We need access to all of the elements of some sequential dataset but **without computing all of those elements in advance and storing them.**

It seems like we’ve seen this before!

Remember our **range** sequence type? A range represents a consecutive, finite sequence of integers. However, **each element of that sequence does not have to be represented explicitly in memory**.

When an element is requested from a range, it is **computed**.

So we can represent very large ranges of integers without using large blocks of memory. Only the end points of the range are stored as part of the range object, and elements are computed on the fly.

for number in range(10000, 1000000000):

An iterator is an object that provides **sequential** access to an underlying sequential dataset.

The iterator abstraction has two components:

A mechanism for retrieving the **next** element.

A mechanism for signaling that the **end of the series** has been reached and no further elements remain.

The advantage of iterators is that the underlying series of data does not have to be represented explicitly in memory.

An iterator provides a mechanism for considering each of a series of values in turn, but all of those elements do not need to be stored simultaneously.

Instead, when the next element is requested from an iterator, that element is **computed on demand i**nstead of being retrieved.

**Iterators do not need to provide access to arbitrary elements of the underlying series**. Instead, **they must only compute the next element of the series,** in order, each time another element is requested.

While not as flexible as accessing arbitrary elements of a sequence (such as random access provided by dictionaries), sequential access to data is often sufficient for some applications.

**15.2. Iterators' Magic Methods**

In Python, iterators are implemented as **classes**.

The iterator class must have two magic methods: **\_\_next\_\_** and **\_\_iter\_\_**.

next(*iterator*[, *default*])

Retrieve the next item from the *iterator* by calling its [\_\_next\_\_()](mk:@MSITStore:C:\Python34\install\Doc\python341.chm::/library/stdtypes.html#iterator.__next__) method. If *default* is given, it is returned if the iterator is exhausted, otherwise [StopIteration](mk:@MSITStore:C:\Python34\install\Doc\python341.chm::/library/exceptions.html#StopIteration) is raised.

When we invoke the \_\_next\_\_ method, we are asking the iterator for the next element of the underlying sequence that it represents. The \_\_next\_\_ method may include any computation to **either retrieve or compute the next element**.

Calls to \_\_next\_\_ make a **mutating** change to the iterator: they advance the position of the iterator.

Hence, multiple calls to \_\_next\_\_ will return sequential (and different) elements of the underlying series.

To signal that the end of the series has been reached, \_\_next\_\_ may **raise a StopIteration exception.**

**The \_\_iter\_\_ method is also required. It simply returns the iterator object.**

**15.3. An Infinite Iterator Example**

As an example, let's implement an odd number infinite iterator:

#-------------------------------------------------------------------------------

# Name: iterators

# Purpose: demonstrate iterators definitions

#

# Author: Rula

#-------------------------------------------------------------------------------

"""

Module to describe various iterators.

"""

class OddNumbers(object):

"""

An iterator for odd positive integers

"""

def \_\_init\_\_(self):

self.current = 1

def \_\_next\_\_(self):

result = self.current

self.current = self.current + 2

return result

def \_\_iter\_\_(self):

return self

Calls to \_\_next\_\_ make a **mutating** change to the iterator: they advance the position of the iterator.

The **instance variable current** stores the current number in the series, and **the \_\_next\_\_ method returns this number and updates it**.

**This \_\_next\_\_ method never raises a StopIteration exception**. **It iterates over the infinite series of odd positive integers.**

Let's test our iterator. We import the iterator class OddNumbers like any other class.

>>> from iterators import OddNumbers

>>> odd = OddNumbers()

>>> odd.\_\_next\_\_()

1

>>> odd.\_\_next\_\_()

3

>>> odd.\_\_next\_\_()

5

>>> odd.\_\_next\_\_()

7

>>> odd.\_\_iter\_\_()

<iterators.OddNumbers object at 0x02D29370>

**\_\_next\_\_() is a magic method. Python calls it whenever we call next() on an iterator instance. So we can also write:**

>>> from iterators import OddNumbers

>>> numbers = OddNumbers()

>>> next(numbers)

1

>>> next(numbers)

3

>> next(numbers)

5

>>> next(numbers)

7

**\_\_iter\_\_() is also a magic method. Python calls it whenever we call iter() on an iterator** instance So we can also write:

>>> from iterators import OddNumbers

>>> numbers = OddNumbers()

>>> next(numbers)

1

>>> iter(numbers)

<iterators.OddNumbers object at 0x02071290>

**15.4. StopIteration**

Sometimes we do not want an infinite iterator: **we want the sequence to end**. To do that, we **raise a StopIteration exception** in the **\_\_next\_\_** method when we reach the limit.

class FiniteOddNumbers(object):

    """

    An iterator for odd positive integers

    Argument:

    limit (int): upper limit on the sequence

    """

    def \_\_init\_\_(self, limit):

        self.current = 1

        self.limit = limit

    def \_\_next\_\_(self):

**if self.current >= self.limit:**

**raise StopIteration**

        else:

            result = self.current

            self.current = self.current + 2

            return result

    def \_\_iter\_\_(self):

        return self

class FiniteOddIterable(object):

    """

    An iterable for odd positive integers

    Argument:

    limit (int): upper limit on the sequence

    """

    def \_\_init\_\_(self, limit):

        self.limit = limit

    def \_\_iter\_\_(self):

**return FiniteOddNumbers(self.limit)**

>>> from iterators import FiniteOddNumbers

>>> odd = FiniteOddNumbers(5)

>>> next(odd)

1

>>>next(odd)

3

>>> next(odd) # self.current is now 5 which is also self.limit

Traceback (most recent call last):

File "<input>", line 1, in <module>

File "C:\Users\Rula\Documents\CS21A\iterators.py", line 44, in \_\_next\_\_

raise StopIteration

StopIteration

>>> next(odd)

Traceback (most recent call last):

File "<input>", line 1, in <module>

File "C:\Users\Rula\Documents\CS21A\iterators.py", line 44, in \_\_next\_\_

raise StopIteration

StopIteration

When the \_\_next\_\_() method raises a StopIteration exception, this signals to the caller that the iteration is exhausted.

**This is not an error**; **it’s a normal condition** that **just means that the iterator has no more values to generate.**

**If the caller is a for loop, it will notice this StopIteration exception and gracefully exit the loop.** (In other words, it will handle the exception.)

This is the key to using iterators in for loops.

**Once an iterator’s next() method raises StopIteration, it will continue to do so on subsequent calls.**

**15.5. What can we do with iterators?**

We can use our iterator object in a for statement :

>>> my\_odd\_sequence = FiniteOddNumbers(10)

>>> for i in my\_odd\_sequence:

... print (i)

...

1

3

5

7

9

We can also write:

>>> for i in FiniteOddNumbers(10):

... print (i)

...

1

3

5

7

9

We can test **for membership with in**. However we can only really do this **if the sequence is finite.**

>>> 3 in FiniteOddNumbers(8)

True

>>> 5 in FiniteOddNumbers(8)

True

>>> 6 in FiniteOddNumbers(8)

False

**And where it makes sense, we can even use sum, min and max**.

>>> sum(FiniteOddNumbers(8))

16

>>> min(FiniteOddNumbers(8))

1

>>> max(FiniteOddNumbers(8))

7

Note that **we cannot index iterators**:

>>> FiniteOddNumbers(10)[1]

Traceback (most recent call last):

File "<input>", line 1, in <module>

TypeError: 'FiniteOddNumbers' object does not support indexing

|  |
| --- |
| **15.6. Iterators' Limitations** |
|  |
|  |
| While very useful, iterators do have some limitations.  The most obvious is that they are **only good for one pass over the data.**  >>> odd = FiniteOddNumbers(5)  >>> next(odd)  1  >>>next(odd)  3  >>> next(odd)  Traceback (most recent call last):    File "<input>", line 1, in <module>    File "C:\Users\Rula\Documents\CS21A\iterators.py", line 44, in \_\_next\_\_      raise StopIteration  StopIteration  Once we reach the end of the series, the odd object becomes useless.  There is no way to reset it.  Iterators also generally require instance variables to keep track of the  progress through the sequence.      def \_\_init\_\_(self, limit):          self.current = 1          self.limit = limit  With complex sequences, it can be difficult for the\_\_next\_\_() method to save its position in the calculation.  Iterators also only allow one 'type' of iteration,  in one direction  We may address this by adding flags to the class but this will clutter the class and decrease readability. |

**15.7. Generators**

**A generator is a special function** that creates, or generates values one at a time.

Generator functions are different from regular functions in that instead of return statements, **they use a yield statement to return elements of a sequence**.

**Generators do not use attributes of an object to track their progress through a series**. Instead, they **control the execution of the generator function**.

You can think of them as resumable functions.

**The generator function runs until the next yield statement is executed** each time the generator's next method is invoked.

The OddNumber iterator can be implemented more compactly using a generator function.

def odd\_generator(limit):

    """ An odd number generator """

    current = 1

    while current < limit:

        yield current

        current = current + 2

Even though we never explicitly define a next method here, Python understands that **when we use the yield statement, we are defining a generator function.**

The first time next is called, execution of the generator function **starts at the beginning and continues until the yield keyword is encountered.**

The function then **pauses and returns the value being yielded**, here **current**.

When **next is called again**, **execution resumes in the generator function where it left off,** on the statement immediately following the yield keyword.

**All local variables in the function will remain intact.** **The value of current is preserved across subsequent calls to next.**

If the yield statement occurs within a loop, execution will continue within the loop as though execution had not been interrupted.

To create a generator object, we just call the generator function like any other function. Note that **this does not actually execute the function code.**

Even though there is no explicit return statement, **a generator function returns a generator object:**

>>> from generators import odd\_generator

>>> numbers = odd\_generator(8)

>>> type(numbers)

<class 'generator'>

We can walk through the generator by calling next() and highlighting the statement(s) execution inside the function:

>>> from generators import odd\_generator

>>> numbers = odd\_generator(8)

>>> next(numbers)

1

The first time next is called, execution of the generator function starts at the beginning and continues until the yield keyword is encountered.

def odd\_generator(limit):

    """ An odd number generator """

    current = 1

    while current < limit:

        yield current    <--

        current = current + 2

The function then pauses and returns the value of current which at this point is 1.

>>> next(numbers)

3

When next() is called again, execution resumes in the generator function where it left off, on the statement immediately following the yield keyword.

def odd\_generator(limit):

    """ An odd number generator """

    current = 1

    while current < limit:

        yield current

        current = current + 2   <--

The value of current is is updated. It is now 3.

def odd\_generator(limit):

    """ An odd number generator """

    current = 1

    while current < limit:   <--

        yield current

        current = current + 2

We are still in the while loop since current is 3 and is less than limit.

def odd\_generator(limit):

    """ An odd number generator """

    current = 1

    while current < limit:

        yield current   <--

        current = current + 2

We get to the yield statement again and the value of current, now 3 is returned.

Calling next again results in the following:

>>> next(numbers)

5

>>> next(numbers)

7

>>> next(numbers)

Traceback (most recent call last):

File "<input>", line 1, in <module>

StopIteration

**When the function has no more statements to be executed, a StopIteration exception is automatically raised.**

**We did not have to add any code to our generator function to raise the exception.**

**15.8. What can we do with generators?**

We can use for loops with generators:

>>> from generators import odd\_generator

>>> for number in odd\_generator(6):

... print(number)

...

1

3

5

And if **we are generating a finite sequence**, we can test for membership with in:

>>> 5 in odd\_generator(6)

True

>>> 4 in odd\_generator(6)

False

>>> 20 in odd\_generator(6)

False

And where it makes sense, we can even use sum, min and max.

>>> sum(odd\_generator(8))

16

>>> min(odd\_generator(8))

1

>>> max(odd\_generator(8))

7

We cannot index generators:

>>> numbers = odd\_generator(8)

>>> numbers [3]

Traceback (most recent call last):

File "<input>", line 1, in <module>

TypeError: 'generator' object is not subscriptable

Generators may be useful for **defining filters or reducing some input.**

def get\_code(file):

    """

    filter out comment lines that start with # from the given file

    yield one non comment line at a time

    """

    for line in file:

        if not line.strip().startswith('#'):

            yield line

In this example, we are only yielding lines read from the file if they don’t start with #.

And this is how we would use the filter generator in my program:

def main():

    with open('wordstats.py', 'r') as program\_file:

        for code\_line in get\_code(program\_file):

            print(code\_line)

Here we are using a for loop with the generator object get\_code(program\_file).

**15.9. Iterables**

We saw how iterators only make a single pass over the elements of an underlying series. After that pass, the iterator will continue to raise a **StopIteration** exception when \_\_next\_\_() is called.

Many applications require iteration over elements multiple times. To solve this problem, we introduce iterables.

An iterable object can be iterated over multiple times.

An iterable produces a fresh instance of an iterator every time something wants to iterate over its data.

To define an iterable class, we need to define one method, \_\_iter\_\_ that **returns a new instance of an iterator when it is invoked.**

In Python, lists, **strings and tuples as well as dictionaries and files are all iterables**.

We may also define our own iterables.

Let’s go back to our iterable class FiniteOddNumbers and define an iterable class from it.

In order to **define an iterable class** FiniteOddIterable **for iterator objects** of the class FiniteOddNumbers, we define the \_\_iter\_\_ method to return a new instance of FiniteOddNumbers.

class FiniteOddNumbers(object):

    """

    An iterator for odd positive integers

    Argument:

    limit (int): upper limit on the sequence

    """

    def \_\_init\_\_(self, limit):

        self.current = 1

        self.limit = limit

    def \_\_next\_\_(self):

        if self.current >= self.limit:f

                raise StopIteration

        else:

            result = self.current

            self.current = self.current + 2

            return result

    def \_\_iter\_\_(self):

        return self

class FiniteOddIterable(object):

    """

    An iterable for odd positive integers

    Argument:

    limit (int): upper limit on the sequence

    """

    def \_\_init\_\_(self, limit):

        self.limit = limit

    def \_\_iter\_\_(self):

**return FiniteOddNumbers(self.limit)**

We can use it as follows:

>>> numbers = FiniteOddIterable(8)

>>> for i in numbers:

... print (i)

...

1

3

5

7

>>> for i in numbers:

... print (i)

...

1

3

5

7

>>> sum(numbers)

16

>>> min(numbers)

1

**Note that there is no next function defined on iterables. Only on the underlying iterator object.**

**16. Comprehensions**

**16.1. List Comprehension – Syntax**

Python supports a concept called list comprehension. It can be used to construct lists in a very natural, easy way.

The general syntax of a list comprehension is as follows:

[element for variable in iterable]

or :

[element for variable in iterable if condition]

**16.2. Initialization with Comprehension**

The simplest example of a list comprehension is the initialization of a list.

Let’s say we have a list of 10 elements and we need to initialize all of these elements to 0. Going back to our comprehension syntax:

[element **for** variable **in** iterable]

We can write:

>>> grades = [0 for i in range (10)]

>>> grades

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The above comprehension is equivalent to:

grades = []

for i in range (10):

grades.append(0)

List comprehensions may also be used to initialize a nested list.

Let’s say we have a list of 5 lists with 3 elements each and we need to initialize all of these elements to 0. We can think of our nested list as consisting of 5 rows and 3 columns.

>>> **one\_row = [0 for column in range (3)]**

>>> **one\_row**

[0, 0, 0]

**>>>items = [one\_row for row in range(5)]**

The two steps above may be combined as:

>>> **items = [[0 for column in range (3)] for row in range(5)]**

[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]

The above comprehension is equivalent to:

items = []

for row in range(5):                      # this is the outer loop

    one\_row =[]

    for column in range(3):           # this is the inner loop

         one\_row.append(0)

    items.append(one\_row)

To visualize our list in rows and columns, we can rewrite it as:

[ [0, 0, 0],

[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]

Note that we could have used i and j or x and y instead of column and row.]

**16.3. Mapping with Comprehension**

A list comprehension provides a compact way of **mapping a list into another list** by applying a function to each of the elements of the list.

Going back to our comprehension syntax:

[element for variable in iterable]

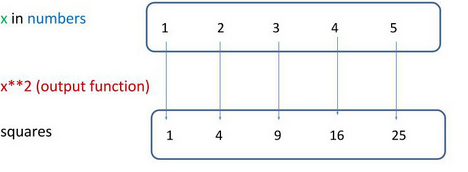
Let's map the numbers list below to a list containing the squares of these numbers:

>>> numbers = [1, 2, 3, 4, 5]

>>> squares= [x\*\*2 for x in numbers]

>>> squares

[1, 4, 9, 16, 25]



numbers is the list we are mapping. The interpreter loops through numbers one element at a time, temporarily assigning the value of each element to the variable x.

Python then applies the function x\*\* 2 and appends that result to the returned list. This function (x\*\*2) is called the output function.

Note that a list comprehension creates a new list. It does NOT change the original list.

>>> numbers

[1, 2, 3, 4, 5]

The above comprehension is equivalent to

squares = []

for x in numbers:

squares.append(x\*\*2)

**16.4. Filtering with Comprehension**

List comprehensions may be used to filter items, producing a result that can be smaller than the original list.

To filter a list, we include the if clause **at the end of the list comprehension**. The expression **after** the if keyword **will be evaluated** for each item in the list. If the expression evaluates to True, the item will be included in the output.

Going back to our comprehension syntax:

[element for variable in iterable if condition]

We can write:

>>> numbers = [1, 2, 3, 4, 5]

>>> even\_numbers = [x for x in numbers if x % 2 == 0]

>>> even\_numbers

[2, 4]

Here we start with the list numbers and we filter it to include only the items that are even in the output list even\_numbers.

The above comprehension is equivalent to:

**even\_numbers = []**

**for x in numbers:**

**if x % 2 == 0:**

**even\_numbers.append(x)**

We can combine mapping and filtering in one comprehension:

>>> numbers = [1, 2, 3, 4, 5]

>>> even\_squares = [x\*\*2 for x in numbers if **x % 2 == 0**]

>>> even\_squares

[4, 16]

Here we start with the list numbers and we map each item to its square (x\*\*2) but then apply a filter to include only the squares of the even numbers.

The above comprehension is equivalent to:

even\_squares = []

for x in numbers:

if x % 2 == 0:

even\_squares.append(x\*\*2)

Going back to the general syntax of list comprehension:

element for variable in iterable]

or :

[element for variable in iterable if condition]

It's important to note that the iterable does not have to be a list or a range object.

We can use a dictionary.

grades = {'Alice': 95, 'Bob': 85, 'Dan': 90, 'Frank': 88}

Suppose that we need a list of all students with a grade of A. We can write:

>>> a\_students = [student for student in grades if grades[student] >= 90 ]

>>> a\_students

['Alice', 'Dan']

The above comprehension is equivalent to:

a\_students = []

for student in grades:

    if grades[student] >= 90:

        a\_students.append(student)

**16.5. Dictionary Comprehension**

A dictionary comprehension is similar to a list comprehension, but it constructs a dictionary instead of a list.

The general syntax of a dictionary comprehension is similar to list comprehension with two differences:

We group the expression using curly braces instead of square brackets.

The expression before the 'for' keyword includes both a key and a value, separated by a colon.

{key:value for variable in iterable}

or :

{key:value for variable in iterable if condition}

**Initializing a dictionary:**

Suppose we need to initialize a dictionary, count, such as each lower case letter has a corresponding initial count value of 0. We could do that by simply typing all the keys and their value (0) as follows:

>>> count = {'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, 'f': 0, 'g': 0, 'h': 0, 'i': 0, 'j': 0, 'k': 0, 'l': 0, 'm': 0, 'n': 0, 'o': 0, 'p': 0, 'q': 0, 'r': 0, 's': 0, 't': 0, 'u': 0, 'v': 0, 'w': 0, 'x': 0, 'y': 0, 'z': 0}

However, this is tedious and error-prone. We could instead use dictionary comprehension to do the same job.

We’ll also use a constant from the string module to get all our lowercase letters.

>>> import string

>>> string.ascii\_lowercase

'abcdefghijklmnopqrstuvwxyz'

>>> count = {letter:0 for letter in string.ascii\_lowercase}

>>> print (count)

{'k': 0, 'j': 0, 'i': 0, 'h': 0, 'o': 0, 'n': 0, 'm': 0, 'l': 0, 'c': 0, 'b': 0, 'a': 0, 'g': 0, 'f': 0, 'e': 0, 'd': 0, 'z': 0, 'y': 0, 'x': 0, 's': 0, 'r': 0, 'q': 0, 'p': 0, 'w': 0, 'v': 0, 'u': 0, 't': 0}

The above comprehension is equivalent to:

count = {}

for letter in string.ascii\_lowercase:

    count[letter] =**0**

**Mapping:**

Now suppose a hypothetical teacher would like to give everyone an extra 2 points on their final grade. We can map an existing dictionary to a new one as follows:

>>> grades = {'Alice': 95, 'Bob': 85, 'Dan': 90, 'Frank': 88}

>>> new\_grades = {student: **grades[student] + 2** for student in grades }

>>> new\_grades

{'Dan': 92, 'Bob': 87, 'Alice': 97, 'Frank': 90}

The above comprehension is equivalent to:

new\_grades = {}

for student in grades:

    new\_grades[student] = **grades[student] + 2**

**Filtering:**

We can also include an if clause in a dictionary comprehension to filter the iterable based on a given expression.

grades = {'Alice': 95, 'Bob': 85, 'Dan': 90, 'Frank': 88}

>>> a\_students = {student: **grades[student] + 2**for student in grades if grades[student] >= 90 }

>>> a\_students

{'Alice': 97, 'Dan': 92}

The above comprehension is equivalent to:

a\_students = {}

for student in grades:

    if grades[student] >= 90:

        a\_students[student] = **grades[student] + 2**

**Swapping Keys and Values**

Dictionary comprehension may be used to swap the keys and values of a dictionary.

>>> address\_book = {'Alice': '555-1234', 'Bob':'555-1111', 'Carol': '555-2222', 'Daniel':'555-3333'}

>>> phone\_lookup = { address\_book[name]: **name** for name in address\_book}

>>> phone\_lookup

{'555-3333': 'Daniel', '555-1111': 'Bob', '555-1234': 'Alice', '555-2222': 'Carol'}

The above comprehension is equivalent to:

phone\_lookup = {}

for name in address\_book:

    value = address\_book[name]

phone\_lookup[value] = **name**

Note that if the values of the dictionary are not unique, the result is not what we may expect:

>>> count = {'the': 300, 'is': 524, 'they': 200, 'an':300}

>>> {count[key]:**key** for key in count}

{200: 'they', 300: '**the**', 524: 'is'}

Because 'the' and 'an' have the same value, only one of them appears in the output dictionary.

**Remember that this swapping only works if the values of the dictionary are immutable, like strings or tuples.**

It will NOT work with a dictionary that contains lists.

>>> my\_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}

>>> {my\_dict[key]:key for key in my\_dict}

Traceback

TypeError: unhashable type: 'list'

**16.6. Set Comprehension**

Sets have their own comprehension syntax as well. It is similar to the syntax for dictionary comprehension and list comprehension.

{element **for** variable **in** iterable}

or :

{element **for** variable **in** iterable**if**condition}

Let’s go back to our example where our task was to identify the students with a grade of A.  We can use the same expression we used with list comprehension but capture the result in a set, instead of a list.

>>> grades = {'Alice': 95, 'Bob': 85, 'Dan': 90, 'Frank': 88}

>>> a\_students = **{**student for student in grades if grades[student] >= 90**}**

>>> a\_students

{'Dan', 'Alice'}

The above comprehension is equivalent to:

a\_students = set()

for student in grades:

    if grades[student] >= 90:

        a\_students.add(student)

**17.1. Module Review**

We have already seen how to create and use Python modules.

In the programming assignments, you have created several modules that you have executed directly.  You have also created modules that only included class definitions.  These modules were not executed directly:  they were imported into another module.

In general, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

**A module allows us to logically organize our Python code.** Grouping related code into a module makes the code easier to understand and use.

We can use any Python source file as a module by executing an **import statement** in some other Python source file or even in the interpreter.

import social

import car

import account

The file name is the module name with the  .py appended.

So when our code is saved in the file  'social.py', the corresponding module name is 'social'.

**17.2. Namespace and Scope**

**A namespace is a collection of identifiers** that belong to a module, to a function, or to a class.

Generally, a namespace holds 'related' things, like all the math functions, or all the date time related behavior.

**Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem.**

To access identifiers belonging to a given module that we have imported, we need to **prefix the identifier with the module name** using the dot notation **modulename.identifiername**.

We refer to this name as a **fully qualified name**.

>>>import math

After importing the math module, we can access the sin function from that module by writing: math.sin()

>>> math.sin(0)

0.0

The **scope of an identifier is the region of program code in which the identifier can be accessed, or used.**

There are three important scopes in Python:

**Local scope refers to identifiers declared within a function.** These identifiers are kept in the namespace that belongs to the function, and each function has its own namespace.

**Global scope refers to all the identifiers declared within the current module.**

**Built-in scope refers to all the identifiers built into Python**— identifiers like len and min that can be used without having to import anything, and are (almost) always available.

Python uses some precedence rules. The same name could occur in more than one of these scopes, but the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope. **Local > global > built-in**

We have also seen a variant of the import statement that **imports names from a module directly into the importing namespace.**

>>> **from** math **import** sin

The sin function has been directly imported into the current namespace and may now be used without being prefixed with math.

>>> sin(0)

0.0

If we prefix it with the module name math, we get an error because the math module itself has not been imported, only the sin function.

>>> math.sin(0)

Traceback (most recent call last):

  File "<string>", line 301, in runcode

  File "<interactive input>", line 1, in <module>

NameError: name 'math' is not defined

The sin function is the only one available.  Other math functions are NOT available.

>>> cos(0)

Traceback (most recent call last):

  File "<string>", line 301, in runcode

  File "<interactive input>", line 1, in <module>

NameError: name 'cos' is not defined

**To figure out what names we have access to at a certain point, after some imports, we use the dir function.**

>>> import account

>>> import math

>>> dir()

['\_\_builtins\_\_', 'account', 'math', 'sys']

>>> from car import Car

>>> dir()

['Car', '\_\_builtins\_\_', 'account', 'math', 'sys']

These are all the names available in our current namespace, at this time.

It is also possible to **import all names from a module into the current namespace** by using the following import statement:

>>> **from math import \***

This provides an easy way to import all the items from a module into the current namespace.  However, this statement **should be used sparingly** as it it can result in confusion if the same identifier names are used in different modules.

>>> dir()

['\_\_builtins\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'sys', 'tan', 'tanh', 'trunc']

**17.3. Module Directory**

We’ve already seen how to use the help function to get help on a module:

>>>import math

>>> help(math)

Help on built-in module math:

NAME

    math

DESCRIPTION

    This module is always available.  It provides access to the

    mathematical functions defined by the C standard.

FUNCTIONS

    acos(...)

        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(..

We may also use the built-in function **dir to get of all the names (variables, functions, classes) that are defined by a given module.**

>>> import math

>>> dir(math)

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

**17.4. The Standard Library**

Python includes a large number of functions, methods and tools and not all of them are available by default. Most of these tools are organized into modules, which make up the **Python standard library.** To use these modules, we have to explicitly import them.

Here are some of the modules included the standard library. We'll cover the highlighted ones in this course.  The Python Standard Library link accessible from  Etudes lists all the available modules.

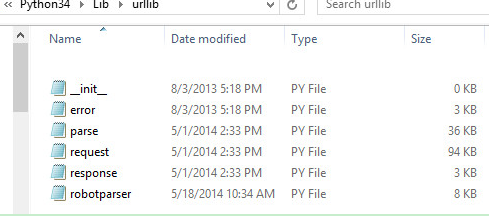
* **os**
* **sys**
* **re**
* math
* **unittest**
* **urllib**
* datetime
* collections
* threading
* array
* **random**
* **tkinter**

There are also many third party modules that we may download and import.

Modules are sometimes grouped into packages.

A **package is a collection of Python modules:** instead of a single file, a package is a directory of Python modules containing an additional \_\_init\_\_.py file.

The urllib package is actually a directory with modules organized as follows:



The module name urllib.request designates the submodule request in the package urllib.

**17.5. The sys Module**

The sys module contains functions and variables that provide access to the environment in which the python interpreter runs.  We'll take a look at some of these variables and functions next.

**The sys.argv variable:**

One of the most useful and widely used variables in the sys module is the sys.argv variable.  It  holds a list of strings read in from the command line when a Python module is run.   These command line arguments can be used to pass information to a program at the time it is invoked.

We specify the command line arguments when we invoke a Python program from the command line or terminal window.

To run a Python module from a Command line window or terminal window, we just type: python followed by the module file name

python helloagain.py

To follow along, copy and save the module helloagain.py available under Resources.

Then open a command line or terminal window.  You can do that from Pycharm by selecting Tools -> Open Terminal or by opening a Command Prompt window (on Windows) or a terminal window on a Mac.

**Navigate to the folder that contains helloagain.py** and type:

python helloagain.py

On Windows, you'll see:

C:\Users\Rula\Documents\CS21A>python helloagain.py

On Mac OS:

rulas-mbp:CS21A rulakhayrallah$ python helloagain.py

If you have both Python 2 and Python 3 installed on your system,  you would need to type:

python3  helloagain.py

With the –m option we specify only the module name:

python –m helloagain

Now we are ready to  **pass arguments to the module from the command line**.

The general format is:

python modulename.py  argument1 argument2 …

For example:

python helloagain.py Bob 5

The arguments are separated by white space.

**If we want to pass an argument with white space in it,** we use double quotes to enclose it.

python helloagain.py "Bob Smith"  5

The **module filename and the arguments** are turned into a list of strings and assigned to the **argv** variable in the sys module.

The length of the list is at least one.  **Even when we don’t explicitly pass any arguments, the module filename is passed to the program.**

Let’s demonstrate how sys.argv works by revisiting our hello friend program.

We would like our program to print a customized hello not just once but a certain number of times.  And we don’t want to prompt the user for their name and the number of times they want the message printed, we just want to let them specify that when they invoke the program with the command line arguments:

python helloagain.py Bob 5

The first thing we need to do is to import sys:

We add the following line at the top of our module:

**import sys**

Inside our module, we now have access to the argument list as sys.argv.

**We can check that the correct number of arguments is passed to our program by checking the length of the list:**len(sys.argv).

Note that the module file name is **ALWAYS** the first item in that list so if we are expecting 2 arguments, the length of our list will be 3.

We can add some more error checking: if the second argument is NOT a number, we'll issue the appropriate error message.

#-------------------------------------------------------------------------------

# Name:        helloagain

# Purpose:     demonstrate command line arguments

#

# Author:      Rula

#

#-------------------------------------------------------------------------------

"""

print a customized hello a given number of times

usage: helloagain.py name number

"""

import sys

def main():

    print ('This is sys.argv:', sys.argv) # for demonstration purposes

    if len(sys.argv) != 3:  #  check for the right number of arguments

        print ('Please try again: helloagain.py name number')

    else:

        name = sys.argv[1]  #  get the name argument

        try:

            number = int(sys.argv[2])  #  get the number argument

        except ValueError:

            print ('Please try again: helloagain.py name number')

        else:   # Print the name the specified number of times

            for i in range (number):

                print ('Hello', name)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Note that **sys.argv is a list**.

* **sys.argv[0] is always the module file name.**

 sys.argv[1] is the first argument specified on the command line after the filename.  Here it is supposed to give us the name.

 sys.argv[2] is the second argument specified on the command line after the filename.  Here it is supposed to give us the number

Let's test our program from the command prompt or the terminal window:

C:\Users\Rula\Documents\CS21A>python helloagain.py Bob  5

This is sys.argv: **['helloagain.py', 'Bob', '5']**

Hello Bob

Hello Bob

Hello Bob

Hello Bob

Hello Bob

sys.argv[0]  is 'helloagain.py'

sys.argv[1] is 'Bob'

sys.argv[2] is '5'

If we specify too many arguments, we get an error message:

C:\Users\Rula\Documents\CS21A>python helloagain.py Bob  5 alice

This is sys.argv: ['helloagain.py', 'Bob', '5', 'alice']

Please try again: helloagain.py name number

And if the second argument is not a number, we get an error message:

C:\Users\Rula\Documents\CS21A>python helloagain.py Bob alice

This is sys.argv: ['helloagain.py', 'Bob', 'alice']

Please try again: helloagain.py name number

**The sys.path variable:**

When we import a module, the Python interpreter searches for it in folders specified in a search path. That search path is stored in the system module sys as the sys.path variable.

To check it out, from the interpreter prompt, we type:

>>>import sys

>>> sys.path

['C:\\Program Files (x86)\\JetBrains\\PyCharm Community Edition 3.4.1\\helpers\\pydev', 'C:\\Windows\\SYSTEM32\\python34.zip', 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages', 'C:\\Users\\Rula\\Documents\\CS21A']

Note that sys.path is just a list.

We can modify it with standard list methods.

>>> **sys.path.insert(0,'c:/documents/Rula/CS21A/grading')**

>>> sys.path

['c:/documents/Rula/CS21A/grading', 'C:\\Program Files (x86)\\JetBrains\\PyCharm Community Edition 3.4.1\\helpers\\pydev', 'C:\\Windows\\SYSTEM32\\python34.zip', 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages', 'C:\\Users\\Rula\\Documents\\CS21A']

**insert(0,…) inserts the folder at the front of the path list.**

>>> sys.path.append('c:/nosuchdir')

>>> sys.path

['c:/documents/Rula/CS21A/grading', 'C:\\Program Files (x86)\\JetBrains\\PyCharm Community Edition 3.4.1\\helpers\\pydev', 'C:\\Windows\\SYSTEM32\\python34.zip', 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'C:\\Python34\\lib\\site-packages', 'C:\\Users\\Rula\\Documents\\CS21A', 'c:/nosuchdir']

Note that we have both forward and backward slashes here**.  Forward slashes are preferable since they don’t have to be escaped in Python strings** (hence the double \\).

append() adds the folder to the end of the path list.

Note that **changes to sys.path are NOT permanent**.  They remain valid only until the current Python process ends or until you exit the current interpreter session.

**sys.exit()**

sys.exit() allows us to exit from the current Python process.

This is implemented by raising the SystemExit exception.

sys.exit takes an optional argument.  It can be an integer giving the exit status (defaulting to zero), or another type of object.

If it is an integer, zero is considered successful termination.

sys.exit("some error message") is a quick way to exit a program when an error occurs.

Since sys.exit()  only raises an exception, it will only exit the process when the exception is not intercepted.

**Standard Input, Output and Error**

sys.stdin, sys.stdout and sys.stderr  are file objects used for standard input, output and errors.

sys.stdin is used for all interactive input

sys.stdout is used for the output of print statements.

sys.stderr is used for error messages.

Sometimes it is useful to redirect the tracebacks from exceptions to a file that will be examined later.

We can do that by adding the following:

**sys.stderr =  open ('errors.txt', 'w')**

The file errors.txt will now contain all the exceptions tracebacks.

**18.1. What are Regular Expressions?**

Regular expressions, also called REs, regexes, or regex patterns are a highly specialized programming language embedded inside Python and made available through the re module.

Regular expressions offer a standardized way of searching, replacing, and parsing text with complex patterns of characters.

Using Regular Expressions, we specify the rules or the pattern for the set of possible strings that we want to match. This set might contain English sentences, e-mail addresses, or anything else we are interested in.

We can then ask questions such as "Is there a match for the pattern anywhere in this string?"

We can also use REs to modify a string or to split it apart in various ways.

**18.2. String Methods or Regular Expressions?**

It is important to remember that in Python, strings have several built-in methods for searching and replacing.

>>> text='Hello! Welcome to CS21A!'

>>> text.find('!')

5

>>> text.replace('!', '.')

'Hello. Welcome to CS21A.'

>>> text.replace('Hello', 'Hi')

'Hi! Welcome to CS21A!'

>>> '.'.join(['filename', 'txt'])

'filename.txt'

The find() method above returns the lowest index in text where the exclamation mark is found.

The replace() method above returns a copy of text where all the occurrences of the exclamation have been replaced by a period

The join() method above returns a concatenation of filename and txt with '.' as the separator.

So before trying to solve a given problem with regular expressions, it is important to check if that task can be accomplished with the string methods. These are simple and easy to use.

When we have to use a lot of different string methods with if statements to handle special cases, it is an indication that we may need to move on to regular expressions.

**18.3. Regular Expressions with Pythex**

Before we look at the available functions in the re module in Python let’s get a hands-on introduction to the regular expression syntax.

One option to quickly test our regular expressions before including them in a program is to use the Pythex website at https://pythex.org/. A Pythex link is also available from the left navigation bar on the Etudes course site.

Note that Pythex does not work with Internet Explorer. Make sure you use Firefox or Chrome when you access it.

So what are we testing with the regular expressions?

The idea is that we have a string and we are looking for a certain pattern in that string.

The pattern may be an email address, of the form: something@something.something

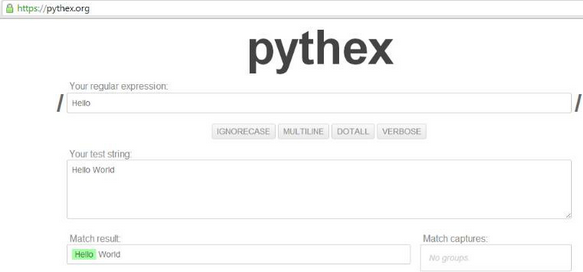
Or a phone number of the form: (nnn) nnn-nnnn

With Pythex we **specify a regular expression** (which is really a pattern) and a string. Pythex displays a match in case there is one.

**Matching characters:**

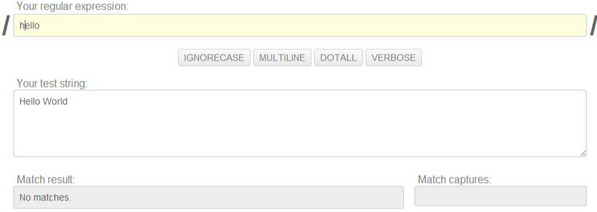
We’ll start with a trivial example: most letters and characters will simply match themselves.

In the screen shot below, 'Hello' is the regular expression and the text we are examining is 'Hello World'. The regular expression 'Hello' matches 'Hello' in 'Hello World' so the match ('Hello') is highlighted in green in the Match result.



Let’s try a few things.  Follow along to get a feel for how Pythex works.

Instead of 'Hello', let’s put in 'hello' for the regular expression.  There is no match.



Now let’s try to match world.  We do get a match because we have specified IGNORECASE.



**Metacharacters:**

Some characters are special metacharacters, and don’t match themselves.

Instead, they indicate that some other thing should be matched, or they affect other parts of the regular expression by repeating them or changing their meaning.

Here’s a complete list of the metacharacters:

. ^ $ \* + ? { } [ ] \ | ( )

We'll take a look at each of them next.

**[] - set of characters**

The square brackets are used to specify a set of characters that we want to match.

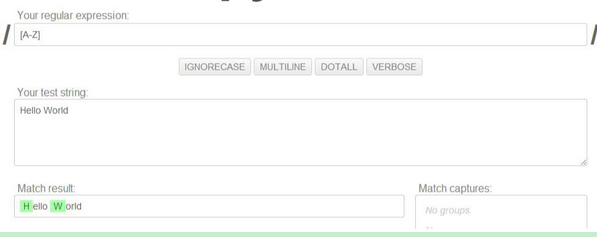
Characters can be listed individually as in [abc].

We can also specify **a range of characters** as two characters separated by a hyphen: [a-c].

So [abc] will match any of the characters a, b, or c; this is the same as [a-c]. If we want to match only lowercase letters, we use [a-z].

[0-9] will match 0, 1, 2, 3, 4, 5, 6, 7,8 and 9.

[A-Z] will match only upper case characters.



Note that **everything within the square bracket, represents one character NOT a sequence of characters.**

**So specifying H[ello] means match:**

**He**

or

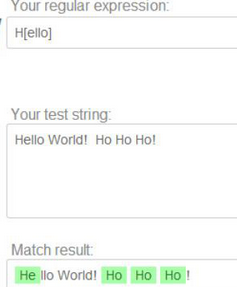
**Hl**

or

**Hl**

or

**Ho**



**Metacharacters are not active inside the square brackets [].**For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside the brackets, it’s just the character $.

We can **match the characters not listed** within the set by **complementing (negating)** the set. This is indicated by including a '^' as the first character of the set. **寻找除了set中因素的其他所有因素**



**. - Any character:**

**The dot . matches any character except a newline character. . 是除换行/n外任何字符**

There is a flag that can be set (DOTALL) so that the dot will match even a newline.

The dot is often used when we want to match 'any character'.

**^ $  - beginning and end:**

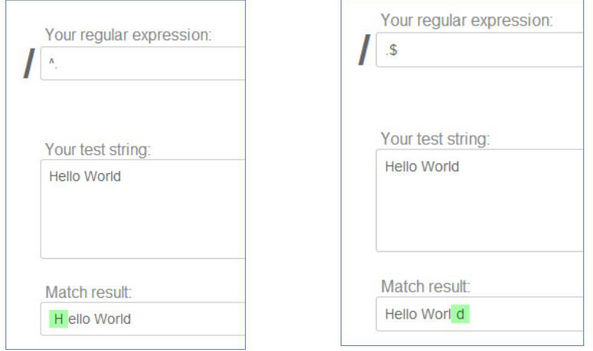
^ matches the beginning of the string or if the MULTILINE flag is set the beginning of each line.

$ matches the end of the string or if the MULTILINE flag is set the end of each line.

**First and Last Characters:**

**^. will match the first character.**

**.$ will match the last character.**



**Repeating Characters: \*, + and ?**

\* indicates that the previous character can be matched **zero or more times. Optional >=0**

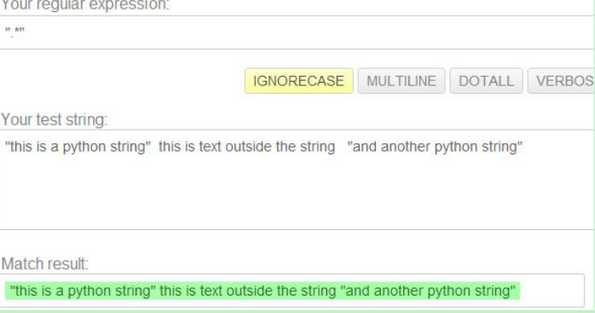
+ indicates that the previous character can be matched **one or more times. At least once >=1**

? indicates that the previous character can be matched **zero or one time.** **Optional or only once 0 or 1** In other words, it indicates an optional character.

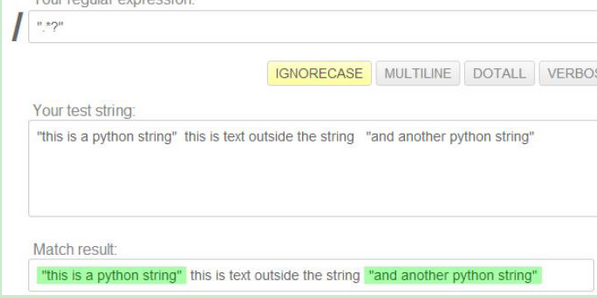
Repetitions such as \* are greedy. When repeating a regular expression, the matching engine will **try to repeat it as many times as possible**.

So o\*ps on the right above matches oooops not ops or oops or ooops.

Sometimes we want the **repetition to be non greedy**. Suppose we are trying to extract double quoted strings from a text**. Using ".\*" does not work** – it just matches everything from the first " to the last ".



To fix that we add a ? after the \* to make the match non-greedy: ".**\*?**" non-greedy



Professor Rula:

As we saw in section 18.3, the question mark means several different things, depending on its context:

When the question mark ? is right after the left parenthesis and followed by : as in  (recogni(**?:**s|z)e), it **disables capturing.**

Sometimes we need the parentheses to indicate precedence but we don’t really want to capture the group.  We can add **?:** inside the parentheses to **disable capturing**.

However the question mark has other uses too. It can be used to indicate an **optional character** as in colou?r.

The question mark may also be used after a \* to make the r**epetition non-greedy**:

Suppose we are trying to extract double quoted strings from a text.  Using ".\*" does not work – it just matches everything from the first " to the last ".

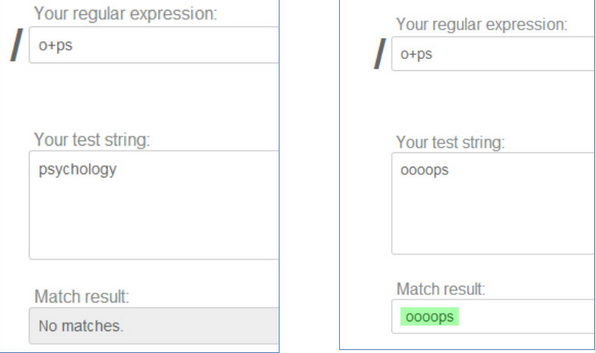
"this is a Python string"  this is text outside the string "and another python string"

To fix that we add a ? after  the \* to make the match non-greedy: ".\*?"

"this is a Python string"  this is text outside the string "and another python string"

**Prefixing with an r merely indicates to the string that backslashes \ should be treated literally and not as escape characters for python**

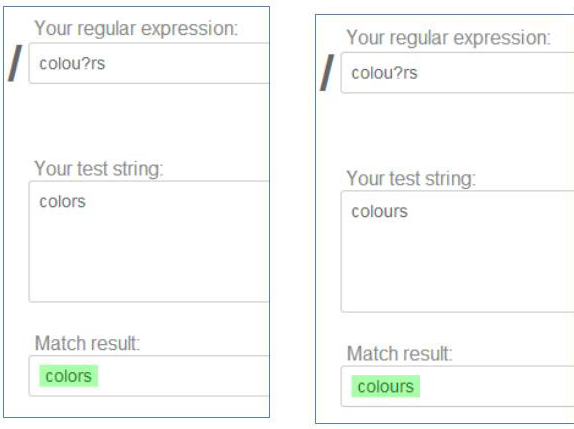
+ indicates that the previous character can be matched **one or more times.**



We’ve seen how to use the question mark character after \* to make the repetition non-greedy.

We can also use the question mark character (without \*) to indicate that the previous character is optional.

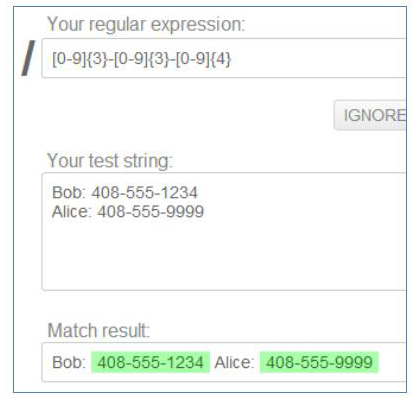
The question mark character, ?, matches either once or zero times. It can be used below to indicate that the 'u' is optional in color (or colour).



**Repeating Characters: {}**

{m} means **repeat the previous character(s) exactly m times**.

[0-9]{3} means repeat the digit [0-9] exactly 3 times.



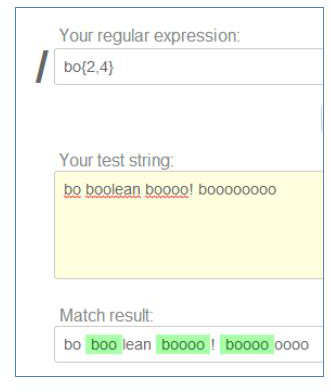
**{m,n} means repeat the previous character(s)** m to n times**.**

**o{2,4}** means repeat the o character **between 2 and 4 times**.

Note that when we write:

bo{2,4}

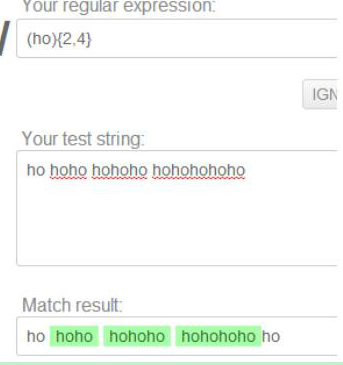
**The {2,4} applies to the o only**. **{m,n}只能作用于前一个字符 最少出现m次,最多出现n次.**



**To repeat a sequence of characters, we use parentheses** as follows:

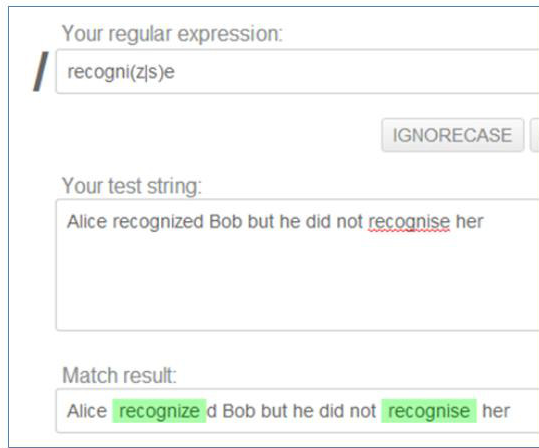
(ho){2,4}

The {2,4} now applies to the 'ho'. () 用来找括号内pattern



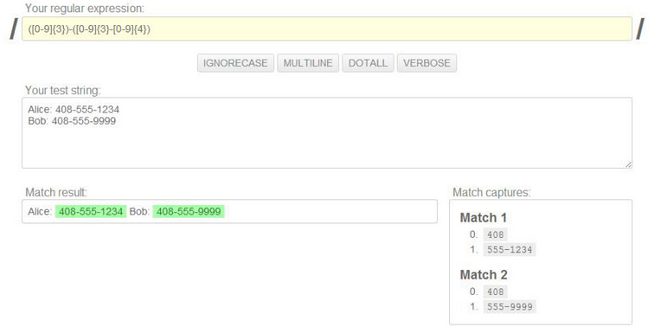
Or: **|**

The vertical bar | indicates a choice.



**Parentheses - ()** The parentheses may be used to group parts of the expression together to indicate precedence like in recogni(z|s)e or (ho){2,4}.

Parentheses also **capture the matched element into a variable (capture group)** that may be used later.



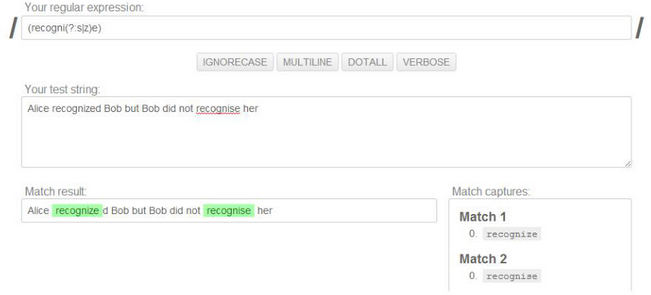
Here we have captured two groups:

([0-9]{3}) is the first one and ([0-9]{3}-[0-9]{4}) is the second

**The two groups appear now under Match captures.**

Sometimes we need the parentheses to indicate precedence 优先级 but we don’t really want to capture the group.

We can add **?:** **inside the parentheses to disable capturing**.



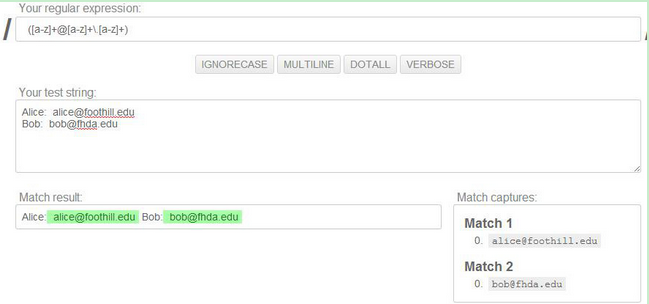
Here we need the parentheses around the (s|z) for precedence but we only want to capture the whole word. So we use **(recogni(?:s|z)e).**

**Escape Character: \**

The backslash may be used as an escape character and it means **treat whatever metacharacter comes after it as a literal**.

For example in an email address pattern, we want to use the dot to actually denote a dot, not any character. We can write:

**([a-z]+@[a-z]+\.[a-z]+)**



The **backslash may also be used to signal a special character.**

Here are some of the more useful special characters:

\d Matches any decimal digit. This is the same as [0-9].

\D Matches any non-digit character. This is the same as [^0-9].

\s Matches any whitespace character \t\n\r\f\v.

\r carriage return 回车

\v vertical tab

\f form feed 换页

\S Matches any non-whitespace character.

\w Matches any word character – that is any alphanumeric character or the underscore. This is the same as [a-zA-Z0-9\_]. 大小写字母 数字 下划线

**\W** Matches any non-word character. This is the same as **[^a-zA-Z0-9\_].**

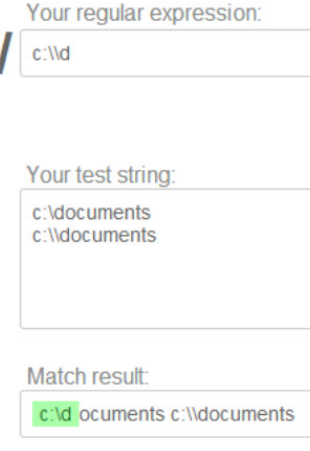
**These sequences can be used inside [].**

For example, **[\s,.]** will match **any whitespace character**, or **','** or **'.'**.

Note that each of these will **match only one character**.

When we actually want to **match a backslash**, we have to escape it – precede it with another backslash: **\\** . The first \ tells the re engine to treat the second \ as a literal (not as a metacharacter).

When we actually want to match a backslash, we have to escape it – precede it with another backslash: \\ . The first \ tells the re engine to treat the second \ as a literal (not as a metacharacter).



So c:\\d will match c:\documents and not c:\\documents.

Suppose that we want to capture email addresses that include special characters such as in the string:

Alice: alice123@foothill.edu

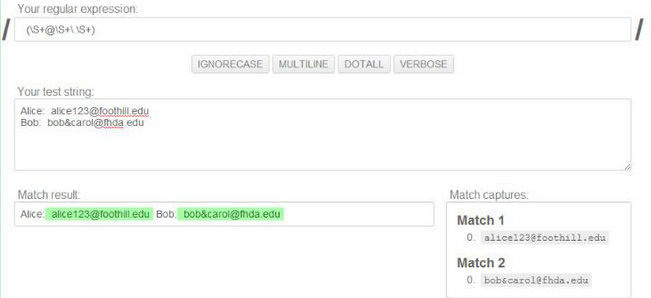
Bob: bob&carol@fhda.edu

The pattern that we have used previously ([a-z]+@[a-z]+\.[a-z]+) will not work.



Using [a-z] excludes the 123 in alice123 and the & in bob&carol.

Instead of [a-z] we can use **\S -** any non whitespace character.



**18.4. The RE Module**

To use regular expressions in Python, we need the re module.

We'll take a look at the following functions available in the re module: match(), search(), findall() and finditer().

To use these functions, we need to import the module.

import re

To call each function, we need to prefix it with the module name:

re.match()

re.search()

re.findall()

re.finditer()

**18.5. Raw Strings**

Before we go on to the specific functions in the re module, we need to talk about raw strings in Python.

Remember that regular expressions use the backslash ('\') for special characters such as \S and \w and to allow metacharacters to be used without invoking their special meaning like \. (for a dot.)

**One \ tells the metacharacter after has no special meaning**

To write one backslash inside a regular expression, we have to write '\\'. The first backslash tells us that the second backslash has no special meaning.

Python also uses the backslash in strings as an escape character: for example we use \n to indicate a new line.

**To write one backslash inside a Python string, we also have to 'escape' it: we write '\\'.** The first backslash tells Python to use the second backslash as is.

As a result if we use Python strings to write a pattern to match one backslash, we have to write '\\\\' as the pattern string:

* The regular expression must be \\ (that’s what we would write in Pythex).
* But then each backslash must be expressed as \\ inside a regular Python string.

The solution is to use Python’s raw string notation for regular expressions. Backslashes are not handled in any special way in a raw string.

**A raw string is a string prefixed with r** such as:

my\_raw\_string = r"hello"

r"\\" is a two-character string containing "\" and "\"

"\\" is a one-character string containing one "\".

In Python it is recommended to use this **raw string notation for all patterns**.

18.6. re.match() and Match Objects

The re.match() function is used to determine if there is **a match at the beginning of the string.**

Syntax: **re.match(pattern, string, flags=0)**

The function **returns a match object on success**, **None on failure**.

Here's an example:

>>> import re

>>> my\_pattern = r'Hello'

>>>print(re.match(my\_pattern, 'Hello World!'))

<\_sre.SRE\_Match object; span=(0, 5), match='Hello'>

>>> print (re.match(my\_pattern, 'Hi Class! Hello World!'))

None

The match function looks for a match at the beginning of the string. Since Hello is not at the beginning in the above example, there is no match and None is returned.

**What’s a match object and how can we access it?**

The group() method may be invoked on the match object to **return the substring that was matched.**

>>> import re

>>> my\_pattern = r'Hello'

>>> my\_match = re.match(my\_pattern, 'Hello World!')

>>> **my\_match.group()**

'Hello'

The start()and end() are also methods defined on the match object: they return the starting and ending index of the match.

The span() method returns both the start and end indexes in a single tuple.

Since the match() function only checks if the pattern specified matches at the start of a string, start() will always return zero when invoked on a match object returned by match.

>>> my\_pattern = r'Hello'

>>> my\_match = re.match(my\_pattern, 'Hello World!')

>>> my\_match.group()

'Hello'

>>> my\_match.start()

0

>>> my\_match.end()

5

>>> my\_match.span()

(0, 5)

**18.7. RE Flags**

The re module functions support a flag parameter. That parameter can take on one or more values.

**re.IGNORECASE**: This flag indicates case-insensitive matching.

>>> my\_pattern = r'hello'

>>> my\_match = re.match(my\_pattern, 'Hello World!', re.IGNORECASE)

>>> my\_match.group()

'Hello'

**re.MULTILINE:** When this flag is specified, the metacharacter '^' matches at the beginning of the string and at the beginning of each line in a multi-line string. Similarly the meta character '$' matches at the end of the string and at the end of each line n a multi-line string. Without this flag , '^' matches only at the beginning of the string, and '$' only at the end of the string.

**re.DOTALL:** This flag makes the '.' metacharacter match any character at all, *including a newline*.

Without this flag, '.' will match anything except a newline.

**re.VERBOSE:** This flag allows us to write regular expressions that look nicer. Whitespace within the pattern is ignored, except inside[] or preceded by a backslash, and, **when a line contains a '#' characters from the leftmost such '#' through the end of the line are ignored.**

To use more than one flag in a single function call, we can use |.

>>> my\_match = re.match(r'h.', 'H\nello', re.IGNORECASE|re.DOTALL)

>>> my\_match.group()

'H\n'

Here we are looking for an h followed by any character (including a new line) and we find it in our string H\nello which is really:

H

Ello

To understand backslashed in python string:

>>> print('\\\\')

\\

>>> print('\\')

\

**18.8. re.search()**

The re.search() function is used to determine if and where the regular expression pattern produces a match at any location in a given string.

Syntax: re.search(pattern, string, flags=0)

The function scans through the string looking for a location where the regular expression pattern produces a match, and **returns a corresponding match object. The function returns None if there is no match.**

Here again we may invoke the group(), start(), end() and span() methods on the match object.

>>> my\_pattern = r'Hello'

>>> my\_match = re.search(my\_pattern, 'Hi Class! Hello Everyone! Hello World!')

>>> my\_match.group()

'Hello'

>>> my\_match.start()

10

>>> my\_match.end()

15

>>> my\_match.span()

(10, 15)

Note that the **search() function only finds the first occurrence of the pattern** in the string.

What if there is no match?

**We should not invoke the group(), start(), end() or span() methods unless we know for sure that the match object exists** and that the search or match functions did not return None. **So usually, we add an if statement as follows:**

my\_pattern = r'Hello'

my\_match = re.search(my\_pattern, 'Hi Class! Hello Everyone! Hello World!')

**if my\_match:** # if m\_match not empty

print('Match found: ', my\_match.group())

**else:**

print('No match')

**18.9. re.findall()**

We noted that search will only find the first occurrence of the pattern in the string.

**To find all the occurrences** we can use the findall() function.

Syntax: **re.findall(pattern, string, flags=0)**

The function finds all non-overlapping substrings where the pattern matches, and **returns them as a list of strings**.

The string is scanned left-to-right, and matches are returned in the order found.

Here's an example:

>>> email\_pattern = r'(\S+@\S+\.\S+)'

>>> text="""Alice: alice123@foothill.edu

… Bob: bob&carol@fhda.edu"""

>>> re.findall(email\_pattern, text)

['alice123@foothill.edu', 'bob&carol@fhda.edu']

**18.10. re.finditer()**

The finditer() function may also be used to find all substrings where the pattern matches.

Syntax: **re.finditer(pattern, string, flags=0)**

The function **returns an iterator yielding match objects** over all **non-overlapping** matches.

This iterator provides sequential access to the match objects. We can iterate over it with a for loop.

The string is scanned left-to-right, and matches are returned in the order found.

Here's an example:

email\_pattern = r'(\S+@\S+\.\S+)'

text='''Alice: alice123@foothill.edu

Bob: bob&carol@fhda.edu'''

for match in re.finditer(email\_pattern, text):

print (match.group())

alice123@foothill.edu

bob&carol@fhda.edu

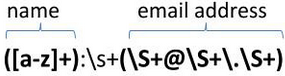
**18.11. Capture Groups**

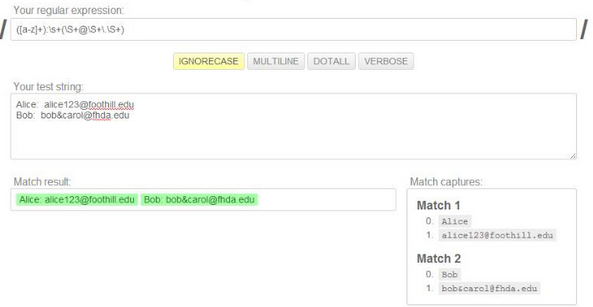
When invoked on the match object the group() method may be used to return one or more **subgroups** of the match.

Remember that parentheses capture the matched element into a group.

Going back to our previous example, let's say we are interested in both the name and the email address. We can write our regular expression as follows:

([a-z]+):\s+(\S+@\S+\.\S+)





We have two capture groups here: the name and the email address.

We also have two matches: one corresponding to Alice and the other corresponding to Bob.

Going back to Python, we would write our code as follows:

my\_template = r'([a-z]+):\s+(\S+@\S+\.\S+)'

addresses = '''Alice: alice123@foothill.edu

Bob: [bob&carol@fhda.edu](mailto:bob&carol@fhda.edu)'''

my\_match = re.search(my\_template,addresses, re.IGNORECASE)

if my\_match:

print(my\_match.group())

Alice: [alice123@foothill.edu](mailto:alice123@foothill.edu)

The group() method, with no arguments (or 0), returns the whole match.

To capture the subgroups separately, we write:

my\_template = r'([a-z]+):\s+(\S+@\S+\.\S+)'

addresses = '''Alice: alice123@foothill.edu

                         Bob:   bob&carol@fhda.edu'''

my\_match = **re.search**(my\_template, addresses,  re.IGNORECASE)

if my\_match:

        print(my\_match.group())

        print ('Name: ',  my\_match.group(1))

        print ('Email address: ', my\_match.group(2))

And here's the corresponding output:

Alice: alice123@foothill.edu

Name:  Alice

Email address:  [alice123@foothill.edu](mailto:alice123@foothill.edu)

group(1) returns the first subgroup.

group(2) returns the second subgroup.

**To capture ALL the names and email addresses**, we may use the **finditer** function and **iterate over the matches** as follows:

my\_template = r'([a-z]+):\s+(\S+@\S+\.\S+)'

addresses = '''Alice: alice123@foothill.edu

                      Bob:   bob&carol@fhda.edu'''

all\_matches = **re.finditer**(my\_template, addresses, re.IGNORECASE)

for my\_match in all\_matches:

        print(my\_match.group())

        print ('Name: ',  my\_match.group(1))

        print ('Email address: ', my\_match.group(2))

And here's the corresponding output:

Alice: alice123@foothill.edu

Name:  Alice

Email address:  alice123@foothill.edu

Bob:   bob&carol@fhda.edu

Name:  Bob

Email address:  [bob&carol@fhda.edu](mailto:bob&carol@fhda.edu)

**18.12. Compiling Regular Expressions**

It is important to note that all the re functions we have encountered so far take a **string parameter as the regular expression pattern**.

It is also possible to 'compile' that pattern into a regular expression object and then invoke the corresponding methods on that object to search a given text.

Here's an example.

>>>compiled\_pattern = re.compile('H')

>>>compiled\_pattern.match('Hello World!')

<\_sre.SRE\_Match object; span=(0, 1), match='H'>

The above is equivalent to the code below:

>>> re.match('H', 'Hello World!')

<\_sre.SRE\_Match object; span=(0, 1), match='H'>

**Compiling the pattern is a common practice in other programming languages to improve performance or because it is the only available way to use regular expressions.**

**However because Python automatically compiles the patterns used with the re functions and stores them in the cache, there is usually no advantage in explicitly compiling our patterns first.**

In this course, we'll use the re functions on the string patterns without compiling them.

Re.SEARCH() 扫描全部string查找pattern 的位置

Re. Match() 只扫描string开始部分对应pattern.

Quiz:

>>> pattern = r'\S'

>>> result = re.findall(pattern, 'Hello there')

Ans: result 结果为 ['H', 'e', 'l', 'l', 'o', 't', 'h', 'e', 'r', 'e']

以上 pattern 可互换做:

pattern=r'[a-z]|[A-Z]'

result

['H', 'e', 'l', 'l', 'o', 't', 'h', 'e', 'r', 'e']

**19. The urllib Package**

**19.1. Web Background and Terminology**

HTTP (short for Hypertext Transfer Protocol) is the message protocol that supports the world wide web. It specifies the format of messages **exchanged between a client, such as a web browser, and a web server.**

Web browsers use the HTTP format to request pages from a web server, and web servers use the HTTP format to send back their responses.

HTTP is a fixed format for communication: it includes specific request and response headers.

An HTTP request also includes a URL.

The URL (Uniform Resource Locator), also known as web address, is the string that constitutes a reference to the 'resource' that is requested.

In most web browsers, the URL of a web page is displayed on top inside an address bar. The following are examples of urls:

http://www.foothill.edu/

https://etudes.org/

The urllib package includes several modules for working with URLs:

urllib.request: for opening and reading URLs

urllib.error: contains the exceptions raised by urllib.request.

urllib.parse for parsing URLs.

urllib.robotparser for parsing robots.txt files

**19.2. urllib.request**

**The urllib.request module** supports **fetching and reading URLs.**

It includes a **urlopen()** function that allows us to 'open' web pages as if they were files.

urlopen() takes the address of the page we want, and returns **a file-like object** that we can just read() to get the full contents of the page.

The read() method on that file-like object always returns bytes, not a string.

To get a string, we need to determine the character encoding and explicitly convert it to a string.

**Each module in the urllib package has to be imported individually**.

>>> import **urllib.request**

We have to use the fully qualified name of urlopen().

>>> url\_file = **urllib.request**.urlopen('http://www.psme.foothill.edu')

One way to look at this: we are the client here. urllib is sending an HTTP request on our behalf to the server at psme.foothill.edu. The server is sending back a response…

And another way to look at it: we are just **opening a remote file-like object.**

Putting it together, we write:

>>> import urllib.request

>>> url\_file = urllib.request.urlopen('http://www.psme.foothill.edu')

>>> type(url\_file)

<class 'http.client.HTTPResponse'>

This is the HTTP response to our request. It is also a file-like object that we can 'read':

>>> page= url\_file.**read()**

We invoke the read() method on that file-like object and save its content into a variable. page is just an arbitrary variable name.

>>> type (page)

<class 'bytes'>

However when we read that file-like object, **we get bytes, not a string**.

>>> print(page)

**b'**<!DOCTYPE html>\n<!--[if IE 6]>\n<html id="ie6" lang="en-US">\n<![endif]-->\n<!--[if IE 7]>\n<html id="ie7" lang="en-US">\n<![endif]-->\n<!--[if IE 8]>\n<html id="ie8" lang="en-US">\n<![endif]-->\n<!--[if !(IE 6) | !(IE 7) | !(IE 8)  ]><!-->\n<html lang="en-US">\n<!--<![endif]-->\n<head>\n<meta **charset="UTF-8**" />\n<meta name="viewport" content="width=device-width" />…

**We can decode the bytes to get a string**. The charset here indicates a UTF-8 encoding.

>>> page.**decode**('UTF-8')

'<!DOCTYPE html>\n<!--[if IE 6]>\n<html id="ie6" lang="en-US">\n<![endif]-->\n<!--[if IE 7]>\n<html id="ie7" lang="en-US">\n<![endif]-->\n<!--[if IE 8]>\n<html id="ie8" lang="en-US">\n<![endif]-->\n<!--[if !(IE 6) | !(IE 7) | !(IE 8)  ]><!-->\n<html lang="en-US">\n<!--<![endif]-->\n<head>\n<meta charset="UTF-8" />\n<meta name="viewport" content="width=device-width" />…

Note that the page we printed is an **HTML page.**

HyperText Markup Language (HTML) is the main markup language for creating web pages that can be displayed in a web browser.

One way to get an idea about HTML is to look at the source for web pages that you commonly use. In most browsers, you can right click on a web page and select View Page Source to see the corresponding HTML source.

**Closing the URL:**

We said that urllib.request.urlopen returns a file-like object.

Once we have read that file, we need to close it.

There are two ways to do that.

We can invoke the **close()** method on the file-like object url-file.

>>> url\_file= urllib.request.urlopen('http://www.psme.foothill.edu')

>>> page= url\_file.read()

>>> url\_file.close()

Or better yet, just like with files, we can use the with statement:

>>>with urllib.request.urlopen('http://www.psme.foothill.edu') as url\_file:

page= url\_file.read()

**And the file-like object is automatically closed when we get out of the with indented block.**

**Capabilities and Limitations:**

The urllib.request module supports some http redirects: when the redirection uses the HTTP 302 response headers. Other redirection schemes are not supported.

It handles some common forms of authentication: basic and digest.

It does not support caching or compression.

**19.3. urllib.parse 解析**

The urllib.parse module defines a standard interface to manipulate URL strings.

It includes functions to:

break up a URL into its components (urlparse).

combine the components back into a URL string.

convert a relative URL to an absolute URL (urljoin).

It supports several URL schemes including http, https, ftp and file.

The general structure of an absolute URL is:

scheme://hostname/path;parameters?query#fragment

Consider the following url:

http://www.foothill.edu/news/newsfmt.php?sr=2&rec\_id=3200

**the scheme identifies the protocol to be used.** The scheme in the example above is http.

**The hostname is also known as the domain (or network location.)** The hostname in the example above is www.foothill.edu.

**The path refers to the specific resource within the host that we want to access**. The path in the example above is **/news/newsfmt.php** .

There are **no parameters** specified here.

**The query is usually a string of name and value pairs** that will be passed to the resource. The query here is **sr=2&rec\_id=3200**

There is no fragment specified here.**The scheme and host components of a URL are not case-sensitive,** but the **path and query string are case-sensitive**. **Usually, the whole URL is specified in lower case.**

**urllib.parse.urlparse() URL 分解函数**

The urllib.parse.urlparse() function breaks up a URL into six components.

**Each component is a string, possibly empty.**

>>> import **urllib.parse**

>>>o=**urllib.parse.urlparse**('http://www.foothill.edu/news/newsfmt.php?sr=2&rec\_id=3200')

>>> type(o)

<class 'urllib.parse.ParseResult'>

>>> print(o)

ParseResult(scheme='http', netloc='www.foothill.edu', path='/news/newsfmt.php', params='', query='sr=2&rec\_id=3200', fragment='')

**urlparse() returns an object with 6 components.** We can access each of these components by its name as follows:

>>> o.scheme

'http'

>>> o.hostname

'www.foothill.edu'

>>> o.path

'/news/newsfmt.php'

>>> o.params

''

>>> o.query

'sr=2&rec\_id=3200'

**Absolute and Relative URLs**

Absolute and relative URLs are similar to absolute and relative file names.

Absolute URLs contain more information but relative URLs are convenient and more portable.

Relative URLs are relative to **a base URL**. **The base URL is the URL we are currently at.**

**We must use absolute URLs when referring to links on different servers.**

**Absolute url: http://www.foothill.edu/counseling/**

**Relative url: counseling/**

**urllib.parse.urljoin()**

The urllib.parse.urljoin() function converts a relative URL to an absolute URL given a 'base URL'.

Let's see how it works with an example. Suppose we are on the Foothill URL:

<http://www.foothill.edu/contact.php>

If we look at the source for that page, we see several 'links'.

Links in HTML are stored as attribute values on various kinds of tags.

Let’s just consider links contained in <a> tags.

<a> tags have an attribute named href that contains the linked URL.

On that page, here are some of the <a> tags we see:

<a href="campuslife/">Campus Life</a>

<a href="http://www.foothill.edu/privacy.php">Privacy Policy</a>

As you can see, one of these URLS is absolute and one is relative.

Let’s try to construct the absolute URLs for each of them.

**Remember that since all these links are from http://www.foothill.edu/contact.php , that URL is my base.**

The syntax of urljoin() is as follows:

**urllib.parse.urljoin(base, url) # 第二个parameter 是 relative URL 短的那个**

>>> import urllib.parse

>>> urllib.parse.urljoin('http://www.foothill.edu/contact.php','campuslife/')

'http://www.foothill.edu/campuslife/'

**urljoin works even if we give it an absolute url. It recognizes it as an absolute and returns it as it is.**

This is convenient because **we don’t have to do  any pre-processing before we call urljoin().**  **We can call it with any url (absolute or relative)** that we encounter and **be certain that we are getting back an absolute url.**

'http://www.foothill.edu/privacy.php'

**19.4. urllib.error**

The urllib.error module defines the exception classes raised by urllib.request.

**urllib.error.URLError**

**urllib.error.HTTPError** – subclass of URLError

**urllib.error.ContentTooShortError** – subclass of URLError

Note that when handling these exceptions, we need to provide their fully **qualified name**.

Example 1:

import urllib.request

import urllib.error

def main():

    url = 'http://nosuchurl.edu'

    try:

        with urllib.request.urlopen(url) as url\_file:

            text= url\_file.read().decode('UTF-8')

    except **urllib.error.URLError** as url\_err:

        print('Error opening url: ', url, url\_err)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Error opening url:  http://nosuchurl.edu <urlopen error [Errno 11001] getaddrinfo failed>

Example 2:

In this example, we combine two steps in our try statement: reading the url and decoding it. To make sure we also catch a decoding exception, we add an except clause as follows:

import urllib.request

import urllib.error

def main():

    url = 'http://nosuchurl.edu'

    try:

        with urllib.request.urlopen(url) as url\_file:

            text= url\_file.read().decode('UTF-8')

    except **urllib.error.URLErro**r as url\_err:

        print('Error opening url: ', url, url\_err)

    except **UnicodeDecodeError** as decode\_err:

        print('Error decoding url: ', url, decode\_err)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

We’ll encounter decoding exceptions in our crawler whenever we have a page that does not use 'UTF-8' encoding.

**19.5. Polite Crawling**

A web crawler is a program that systematically browses the web to extract information. Search engines use web crawlers to update their indexes of web content. Web crawlers are also used for scraping the web for contact information (spamming) or online price comparison or news compilation, etc…

Web crawlers can disrupt networks and Web servers. If a crawler sends multiple requests to one server and downloads large files, the server’s performance will be affected, especially if there are several crawlers.

A partial solution to these problems is the **robots exclusion protocol**, also known as the robots.txt protocol. It is a standard for administrators to indicate which parts of their web servers should not be accessed by crawlers.

A polite web crawler tries to comply with the robots exclusion protocol and not crawl web sites if rules in the server's robots.txt file disallow crawling.

The robots.txt file contains the instructions in a specific format.

Let's take a look at the robots.txt file for www.foothill.edu.

http://www.foothill.edu/robots.txt

User-agent: \*

Disallow: /cms/

Disallow: /cgi/

Disallow: /cgi-bin/

Disallow: /support/

User-agent: dotbot

Disallow: /

**User-agent is used to identify the crawler**.

**dotbot is an e-commerce crawler and is asked not to crawl foothill.edu**

**All other crawlers (\*) are asked not to crawl cms, cgi, cgi-bin and support.**

Here's the robots.txt file for facebook.com.

http://facebook.com/robots.txt

User-agent: Googlebot

Disallow: /ac.php

Disallow: /ae.php

Disallow: /ajax/

Disallow: /album.php

Disallow: /ap.php

Disallow: /autologin.php

…

User-agent: msnbot

Disallow: /ac.php

….

User-agent: \*

Disallow: /

**19.6. urllib.robotparser**

The **urllib.robotparser** module **answers questions about whether or not a particular user agent can (politely) fetch a URL on a given web site.**

Here's an example:

We first import the module:

>>> import **urllib.robotparser**

**Instantiate a RobotFileParser object:**

>>> rp = urllib.robotparser.RobotFileParser()

**Set the URL to the robots.txt file corresponding to the domain:**

>>> rp.set\_url("http://foothill.edu/robots.txt")

**Read the robots.txt file:**

**>>>** rp.read()

**Answer 'can fetch' questions for various user agents:**

>>> rp.can\_fetch("\*", "http://foothill.edu/campuslife/")

True

>>> rp.can\_fetch("\*","http://www.foothill.edu/support/")

False

**19.7. A Simple Web Crawler**

Your task this week is to implement a simple web crawler.

A web crawler is a program that **systematically browses the web** to **extract information.**

Search engines use web crawlers to update their indexes of web content.

Web crawlers are also used for scraping the web for contact information (spamming) or online price comparison or news compilation, etc…

Web crawlers **start their crawling from some initial url called the seed**.

Then **they go on to all the web pages that are directly or indirectly linked to from that seed.**

The overall algorithm is as follows:

* Select a URL that has not yet been crawled
* Download the  page corresponding to that URL
* Harvest the information you need from that page.  This may include all the words referenced on that page, the prices, the email addresses, etc… That information is then saved for later retrieval.
* Extract the links referenced from that page and for each of these links:
* Add the URL to the set of URLs that still need to be crawled
* Repeat until there are no more URLs to crawl

For a search engine crawler for instance, we build an index containing words and references to web pages that contain them.

In this week's assignment, we will skip the word indexing part (you implemented something similar in a previous assignment – wordstats.)

We will not harvest any information from the web pages other than the urls: **we’ll just crawl them and report back on our crawling path.**

So our **simplified algorithm** will look like this:

Select a URL that has not yet been crawled

Download the page corresponding to that URL

Extract the links referenced from that page and for each of these links:

Add the URL to the set of URLs that still need to be crawled

Repeat until there are no more URLs to crawl

Let's look at each step in the simplified algorithm and map it to the corresponding function in crawler.

**Select a URL that has not yet been crawled:**

This step has been implemented for you in the function crawl. The url is selected randomly (pop() gets a random item from the set and removes it):

**current\_url= urls\_tocrawl.pop()**

**Download the page corresponding to that URL:**

You'll implement this step in the function get\_page(). You'll use urlurllib.request.urlopen() to **open** the ulr then you'll **read and decode it**. Make sure that you handle both URL opening errors and decoding errors and generate the appropriate messsage for each. **The function should return an empty string in case of error**.

def get\_page(url):

    # TO DO:

    # get\_page takes an absolute url as input parameter

    # and returns **a string** that contains the web page pointed to by that url.

    # Assume the web page uses utf-8 encoding.

    # **If there is an error opening the url or decoding the content,**

    # **print an error message and return an empty string**.

    # Use the with construct to open the url.

**Extract the links referenced from that page.**

You will implement this step in the function extract\_links() outlined here and in the template provided under Resource.

def extract\_links(base\_url, page):

    """

    extract the links contained in the page at the base\_url

    Parameters:

    base\_url (string): the url we are currently crawling - web address

    page(string):  the content of that url - html

    Returns:

**A set of absolute urls (set of strings**) - These are all the urls extracted

        from the current url and converted to absolute urls.

    """

    # TO DO:

    # write the code for extract\_links.

    # 1. Initialize an empty set for the urls

    # 2. **use a re pattern to extract the urls from the html file**

    # 3. make sure you extract ALL the URLs on that page

    # 4. convert each link to an absolute url (hint: **urllib.parse.urljoin())**

    # 5. **call the function ok\_to\_crawl** to check if you are allowed

    #    to crawl that absolute url

    # 6. If that url is ok\_to\_crawl, add it to the set of urls

    #    found on that page.

    # 7. Return the set of urls

You'll use a regular expression to extract the linked URLs from the HTML source document read by the crawler.

Before using the regular expression in the crawler, it will be helpful to **test it with  Pythex.**

You'll also have to **capture the URL referenced in a capture group** (by enclosing it in parentheses.)

Here's some background to guide you in this step.

Links in HTML documents may be stored as attribute values on various kinds of HTML tags.  You only need to extract linked URLs contained in <a> tags.

<a> tags have an attribute named href that contains the URL.

The general simplified  format is:

<a href="url">url description</a>

We'll assume the URLs are delimited with **double-quotes**.

Make sure you handle extra white space characters (space or new line) around the = sign and between the a and href:

<a    href =  "url" >url description</a>

To test your regular expression, make sure it extracts all 6 links from the following document.  Copy and paste it into Pythex and test it with your pattern:

<!DOCTYPE html>

<html>

<head>

<title>RE Test</title>

</head>

<body>

<h1>Testing Regular Expressions</h1>

<p>Extract the links from this document</p>

<div>

<a href="http://www.krauseinnovationcenter.org/">Krause Center</a>

</div>

<div>

<a href=

"counseling/">Counseling</a>

<br>

<a href ="aid/">Financial Aid</a> <br><a href = "library/">Library</a>

<br>

<a  href ="http://books.foothill.edu/home.aspx"  >Bookstore</a>

<br><a

href="services.php">Online Services</a>

</div>

</body>

</html>

Note that while your regular expression pattern will include <a and href, the captured url has to consist of the blue text above only.

20. Graphical User Interfaces

**20.1. What's a GUI?**

A Graphical User Interface is an interface that lets us use icons, toolbars, buttons and menus to achieve tasks. We point, click, drag, select and the corresponding command is executed.

In contrast, a command line interface is an interface that is text based: we type in our commands one line at a time.

Python provides several third party frameworks for developing graphical user interfaces. For a complete list of these available options, visit: https://wiki.python.org/moin/GuiProgramming

Tkinter is the standard GUI library for Python. It comes bundled with Python. It offers a fast and easy way to create GUI applications. It provides a powerful object-oriented interface to the Tk GUI toolkit.

IDLE is a Tkinter GUI.

In this course we'll use Tkinter to illustrate GUI programming in Python.

The Tkinter reference is available from the left navigation bar on Etudes.

**20.2. Event Driven Programming**

GUI applications are **event driven**. They sit and wait for events to happen.

Events can come from various sources, including key presses and mouse clicks by the user.

When a significant event occurs, they handle it then go back to waiting for the next event.

The following entities are associated with a given event:

**The event type** (or **event name**) describes the event: is it a mouse click, a character input?

**The event target is** the object on which the event happened: which specific item (or widget) was clicked on?

**The event handler** or **event listener** is the **function or method** that is invoked to respond to the event.

**20.3. Our First GUI Application**

Let’s create our first GUI application with tkinter.

Here's what we need to do:

1. Import the tkinter module.
2. Create the GUI application main window.
3. Add some widgets to the GUI application.
4. Enter the main event loop and wait for events triggered by the user.

Our First GUI Application:

# import the tkinter module - step 1

import tkinter

def main():

    # create the GUI application main window - step 2

    root = tkinter.Tk()

    # enter the main event loop and wait for events - step 4

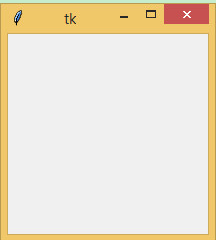
    root.mainloop()

 if \_\_name\_\_ == '\_\_main\_\_':

    main()

Note that we have not created any widgets here. We’ll come back to that later…

We can run our application as it it. We get the following:



**class tkinter.Tk(**screenName=None, baseName=None, className='Tk', useTk=1)

The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

**tkinter.Tcl**(screenName=None, baseName=None, className='Tk', useTk=0)

The Tcl() function is a factory function which creates an object much like that created by the Tk class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn’t want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the Tcl() object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

Our first GUI Application doesn't do much: it opens a window and waits…

To stop the program, we need to close the window.

Before moving on to a more interesting application, let's take a closer look at each of the steps involved above:

**Importing tkinter:**

Using 'import tkinter' to import the module implies that to access tkinter identifiers, we'll need to prefix them with tkinter. So **when we write tkinter.Tk(), we are instantiating an object of the Tk class**; the Tk class is defined in the tkinter module.

In a very large GUI application, this prefixing can get tedious and confusing. In this case, it is OK to import everything into the application's namespace as follows:

**from tkinter import \***

**Creating the main window:**

root = tkinter.Tk()

Here we have created a top level window for our application: it is **an instance of the class tkinter.Tk**.

By convention, the top level window is usually named root. **We should only create one root window for each program and it must be created before any other widgets**.

**Entering the Event Loop:**

root.mainloop()

Here we are invoking the mainloop method on the root object.

As the mainloop runs, it waits for events to happen.

If an event occurs, then it is handled by the corresponding handler if such a handler exists and the loop continues running, waiting for the next event.

The loop continues to execute until the root window is closed.

**20.4. General GUI Design**

When we develop a GUI application, there are some general tasks that we need to accomplish:

1. We need to specify what the application will look like: we do that by creating **different visual components.** In tkinter, these components are called **widgets**.
2. We need to specify what the application will do: we do that by writing methods and functions that perform certain tasks. These **methods and functions are the event handlers**, also called **callbacks. 回调函数**
3. We need to **associate the looking with the doing**: we do that by **associating specific events on widgets with the event handlers we have written**.
4. We need to write code that sits and waits for input from the user. We've seen how to do that in the previous section.

We'll now go on to see how to achieve the first three subtasks above.

**20.5. Widgets**

We specify how we want a GUI to look by describing the widgets that we want it to display, and their spatial relationships 空间关系 (whether one widget is above or below, or to the right or left, of other widgets).

**Tkinter widgets include labels,标签 frames, buttons, entries , menus, canvases 画布**and more. We’ll cover the most useful ones.

We can instantiate all widgets using the same general syntax:

**WidgetClass(parent, option = value, ...)**

This creates an instance of the WidgetClass, with the given parent, using the specified options.

All options have default values, so in the simplest case, we only have to specify the parent widget.

Every widget has a parent widget. However if we do not specify the parent, tkinter uses the most recently created root window as the parent.

Let's start by creating a **Label widget**.

**A Label widget can display text or an icon or some other image**. **It is used to label other widgets**. Unlike most other widgets, **labels are not interactive**.

A label widget is an instance of the Label class. Following the general syntax for instantiating widgets, we can create a label as follows:

**my\_label = tkinter.Label(parent, option, ...)**

And here's a more specific example:

# import the tkinter module

import tkinter

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # instantiate a Label widget with root as the parent widget

    # use the text option to specify the text to display

    hello = tkinter.Label(root, text = 'Hello World!' )

    # invoke the pack method on the widget

    hello.pack()

    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

We run our program and we get our label:



Here again, to exit the program, we need to close the window.

Before moving on to other widgets, let's clarify some of the concepts introduced in the example above.

**Parent**: when we create a widget, we associate it with a parent.

The statement tkinter.Label(my\_parent) creates an instance of the Label class and associates that label instance with its parent. **The concept of parent is used in determining the lifetime of a widget.** When a parent component (such as the root) is closed, the parent knows who its children are, and can close them/destroy them before destroying itself.  
**Packing:** **the pack method is invoked to ensure the widget is visible**. If we don't pack a widget, we will not see it. We could have used the **grid** (or the **place**) method instead. We’ll come back to that later.

The **Frame widget** lets us **group widgets**. It works like a **containe**r, and it is responsible for **arranging the position of other widgets**.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets.

A frame widget is an instance of the Frame class:

A frame widget is an instance of the Frame class:

**my\_frame = tkinter.Frame(parent, option, ...)**

Let's add two frames to our app:

# import the tkinter module

import tkinter

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # instantiate a Label widget with root as the parent widget

    # use the text option to specify which text to display

    hello = tkinter.Label(root, text = 'Hello World!' )

    # invoke the pack method on the widget to display it

    hello.pack()

    # we create a top frame

**top\_frame** = tkinter.Frame(root)

    top\_frame.pack()

    # and a bottom frame

**bottom\_frame** = tkinter.Frame(root)

    # side is an optional parameter of the pack method.  It defaults to TOP.

    bottom\_frame.pack(side = tkinter.BOTTOM)

    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Here we have created 2 frames. When we run the program, the window looks exactly the same, nothing has changed. The frames are empty containers.



The **Button widget** is used to add buttons in a GUI application. These buttons can display text or images. They can also be 'clicked'.

We usually attach a function or a method to a button. That function or method is then called whenever the button is clicked.

A button widget is an instance of the Button class:

**my\_button = tkinter.Button(parent, option, ...)**

Buttons, like other widgets have a variety of options to control their size, their color, the text that they display, how their borders look, and so on. Note that some options such as the background or foreground colors on buttons are NOT supported on MAC OS.

In this example, we will set just one options: the text.

# import the tkinter module

import tkinter

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # instantiate a Label widget with root as the parent widget

    # use the text opetion to specify which text to display

    hello = tkinter.Label(root, text = 'Hello World!' )

    # invoke the pack method on the widget

    hello.pack()

    # we create a top frame

    top\_frame = tkinter.Frame(root)

    top\_frame.pack()

    # and a bottom frame

    bottom\_frame = tkinter.Frame(root)

    # side is an optional parameter of the pack method.  It defaults to TOP.

    bottom\_frame.pack(side = tkinter.BOTTOM)

    # create a STOP button in the top frame

**stop\_button = tkinter.Button(top\_frame, text='STOP')**

    stop\_button.pack()

    # create a GO button in the bottom frame

**go\_button = tkinter.Button(bottom\_frame, text='GO')**

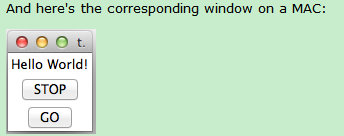
    go\_button.pack()

    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()



Note that unlike the labels, the **buttons are clickable**.

The **Canvas widget 画布** allows us to **create a rectangular area intended for drawing pictures** or other layouts.

**A canvas widget is an instance of the Canvas class:**

**my\_canvas = tkinter.Canvas(parent, option, ...)**

Here's an example using the canvas widget.

# import the tkinter module

import tkinter

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # create a label

    title = tkinter.Label(root, text = "Let's Draw!")

    title.pack()

    # instantiate a Canvas widget with root as the parent widget

**canvas= tkinter.Canvas(root, background = 'green')**

    # draw a blue rectangle on the canvas

    canvas.create\_rectangle(50, 50, 150, 100, fill = 'blue')

    # and two red circles

    canvas.create\_oval(50, 100, 75, 125, fill = 'red')

    canvas.create\_oval(125, 100, 150, 125, fill = 'red')

    canvas.pack()

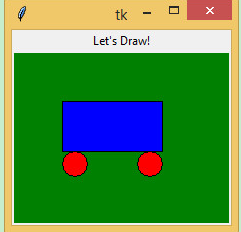
    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

And here's what we get:

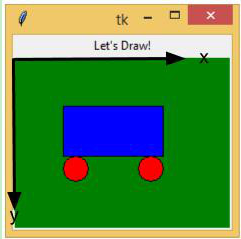


For creating rectangle object on canvas:

**id = C.create\_rectangle(x0, y0, x1, y1, option, ...)**

Let's take a closer look at the coordinate system used here.

The origin of the coordinate system **(x,y) is at the upper left corner of canvas**, with the x coordinate increasing toward the right, and the y coordinate increasing toward the bottom:

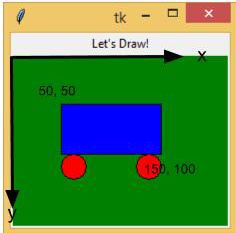


The base unit is the pixel, with the top left pixel having coordinates (0,0). We can specify the coordinates as integers to denote pixels. We can also specify inches or centimeters as follows:

canvas.create\_rectangle('1i','1i', '3i', '3i', fill = 'magenta')

canvas.create\_rectangle('1c','1c', '3c', '3c', fill = 'magenta')

To draw a rectangle we specify two points: the top left corner and the bottom right corner.



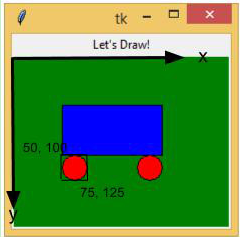
canvas.create\_rectangle(**50, 50, 150, 100**, fill = 'blue')

**50, 50** is the location of the top left corner

**150, 100** is the location of the pixel just outside of the bottom right corner.

All coordinates are relative to the canvas (not the root window.)

To draw an oval (or a circle) we fit it into a rectangle first and then we specify the top left corner and the bottom right corner of that rectangle.



canvas.create\_oval(**50, 100, 75, 125,** fill = 'red')

Similarly to draw a line on a canvas, we specify the coordinates of the endpoints:

canvas.create\_line(50, 50, 150, 200)

**20.6. Geometry Managers**

**When we create a widget, it does not appear in the window until we register it with a geometry manager.**

There are 3 geometry managers available in tkinter.

1. The **pack geometry manager** creates a layout by 'packing' the widgets into a parent widget, treating them as **rectangular blocks**.
2. The **grid g**eometry manager creates table-like layouts, organizing the widgets in **a 2-dimensional grid**. **Row and Columns**
3. The **place geometry** manager allows us to explicitly place a widget in a given position. It is seldom needed.

All widgets have the following methods corresponding to the 3 geometry managers:

* pack()
* grid()
* place()

We'll illustrate the use of the grid() geometry manager with a simple example.

The grid manager treats every window or frame as a table of rows and columns.

**The width of each column is the width of the widest cell in that column.**

**The height of each row is the height of the largest cell in that row.**

# import the tkinter module

import tkinter

# this is our color dictionary

COLORS = {'red': 'rouge', 'blue': 'bleu', 'green': 'vert',

          'yellow': 'jaune', 'black': 'noir', 'white': 'blanc'}

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # set the title for the window

    root.title("Colors")

    # add a label

    table\_label = tkinter.Label(root, text = "Color Review" )

    # register it with a geometry manager

    table\_label.grid()

    # instantiate a frame for our table

    table = tkinter.Frame(root)

    # register it with a geometry manager

    table.grid()

    row\_count = 0

    # loop through the colors in our dictionary

    for color in COLORS:

        # instantiate a label for the English name/ the key name

        color\_english = tkinter.**Label**(table, text = color)

        # instantiate a label for the French name / the key value

        color\_french = tkinter.**Label**(table, text=COLORS[color])

        # instantiate a label to show the color

        color\_swatch = tkinter.**Label**(table, background = color, width = 12)

        # each label goes in a different column - same row.

        color\_english.grid(row = row\_count, column = 0)

        color\_french.grid(row = row\_count,column = 1)

        color\_swatch.grid(row = row\_count,column = 2)

        row\_count += 1

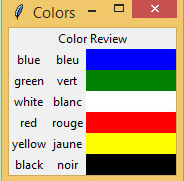
    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

   main()

 And here's our window. There are 6 rows and 3 columns in the table frame.



Note that the grid placement is relative to the parent, which in this case is the table frame, not the root window.

**20.7. Events and Handlers**

We have seen how to display various components in our GUI window but we have not seen how to capture the user input and initiate an action based on that input.

Several widgets accept user input and there are **two ways** to associate actions with events on widgets.

Some widgets such as buttons have a **command option** that lets us specify a function or a method, called a **handler**. The handler will be called whenever the user clicks that button.

Let's illustrate that option with a generic board game GUI with a START and QUIT buttons.

#-------------------------------------------------------------------------------

# Name:        game

# Purpose:     Implement a general board game

#

# Author:      Rula Khayrallah

#-------------------------------------------------------------------------------

"""

Module to implement a generic board GUI game app

"""

import tkinter

class Game(object):

    """

    GUI Game class to support a general purpose 8 x 8 board game

    Attributes:

    frame:  (tkinter.Frame) container for the game board and buttons

    parent: (tkinter.Tk) the root window, parent of the frame

    """

    # we specify a class variable for the tile\_size

    tile\_size = 50;

    def \_\_init\_\_(self, parent):

        parent.title('CS 21A Board Game')

        self.parent = parent

        self.initialize\_board()

    def initialize\_board(self):

        """Initialize the game board """

        # create the frame

        self.frame=tkinter.Frame(self.parent)

        # register it with a geometry manager

        self.frame.grid()

        # create a START button and associate it with the start method

        start\_button = tkinter.Button(self.frame, text = 'START',

                                        width = 20,

**command = self.start)**

        # register it with a geometry manager

        start\_button.grid()

        # create a QUIT button and associate it with the quit method

        quit\_button = tkinter.Button(self.frame, text = 'QUIT',

                                        width = 20,

**command = self.quit)**

        # register it with a geometry manager

        quit\_button.grid()

        # create a canvas to draw our board

        self.canvas = tkinter.Canvas(self.frame,

                                     width= self.tile\_size \* 8,

                                     height= self.tile\_size\* 8 )

        # register it with a geometry manager

        self.canvas.grid()

        # draw the tiles on the canvas

        for row in range(8):

            for column in range(8):

                if (row + column)% 2 == 0:

                    color = 'black'

                else:

                    color = 'red'

                self.canvas.create\_rectangle(self.tile\_size \* column ,

                                     self.tile\_size \* row,

                                     self.tile\_size \* (column + 1) ,

                                     self.tile\_size \* (row + 1),

                                     fill = color)

    def start(self):

        """ this method is called when the START button is pressed"""

        print('starting the game')

    def quit(self):

        """ this method is called when the QUIT button is pressed"""

        print('quitting the game')

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # instantiate our Game object

    my\_game = Game(root)

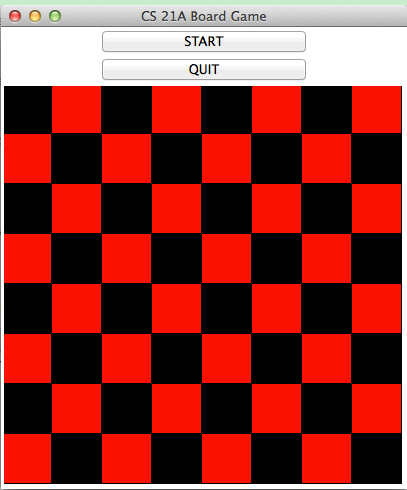
    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

And here's our game window:



If we press on the START and QUIT buttons, we get the following output in the console:

starting the game

quitting the game

The command option does not allow us to pass any arguments to the function or method. That's one reason why using object oriented design is advantageous. The method has at least access to the underlying object.

**Binding** is a more general mechanism that **allows the application to respond to many more kinds of inputs: the press or release of any keyboard key or mouse button; movement of the mouse into, around, or out of a widget; and many other events**.

For each widget, we bind functions and methods to events: if an event matching the event description occurs in the widget, the given function or method is invoked.

The function or method (handler) is passed an event object that contains more details about the event.

The general syntax is:

**widget.bind(event, handler)**

The process of binding involves **three different entities**:

1. a widget such as a **button, a canvas, a frame**
2. **a type of event** such as **a click** of the left mouse button, or **a press** of a certain key on the keyboard.
3. **an event handler** which is **a function or method**.

Here are **some** of the most common **event types** and their meaning.  We'll illustrate their use with a simple drawing application in the next section.

<Button-1> - The **leftmost mouse button** is pressed over the widget.

<Button-2> - The **middle mouse button** is pressed over the widget.  <Button-2> also corresponds to the secondary click on the MAC trackpad.

<Button-3> - The **rightmost mouse button** is pressed over the widget.

<Double-Button-1> - The leftmost mouse button is double clicked over the widget.

<Double-Button-2> - The middle mouse button is double clicked over the widget.

<Double-Button-3> - The rightmost mouse button is pressed over the widget.

<B1-Motion> - The mouse is **moved, with the leftmost mouse button held down**.

<B2-Motion> - The mouse is moved, with the middle mouse button held down.

<B3-Motion> - The mouse is moved, with the rightmost mouse button held down.

<ButtonRelease-1> The leftmost mouse button was released.

<ButtonRelease-2> The middle mouse button was released.

<ButtonRelease-3> The rightmost mouse button was released.

<Enter> **The mouse pointer entered the widget** It does NOT mean that the user pressed the Enter key.

<Leave> **The mouse pointer left the widget**.

Character Events:  most printable characters can be used as is to denote an event.  键盘敲入任何符号 符号事件

a - The user typed an 'a'.

**The Event Object:**

**The event handler receives one argument: the event object.**

That event object is a Python object, with a number of attributes describing the event. Here are some of the event object attributes:

**widget:** the widget which generated this event.   This is a valid Tkinter widget instance, not a name.

**x, y:** the current mouse position, in pixels.

**char:** the character code (keyboard events only), as a string.

**20.8. A Simple Drawing Application**

Let's create a simple drawing app to illustrate the bind method, event types and the use of the event object.

#-------------------------------------------------------------------------------

# Name:        draw

# Purpose:     Implement a simple drawing application

#

# Author:      Rula Khayrallah

#-------------------------------------------------------------------------------

"""

Module to implement a simple drawing app

"""

import tkinter

class DrawApp(object):

    """

    class to support a simple drawing app

    Attributes:

    canvas:  (tkinter.Canvas) our drawing canvas

    parent: (tkinter.Tk) the root window, parent of the frame

    """

    def \_\_init\_\_(self, parent):

        parent.title('CS 21A Drawing App')

        self.parent = parent

        # create the canvas

        self.canvas=tkinter.Canvas(self.parent, width = 300, height = 300)

        # register it with a geometry manager

        self.canvas.grid()

        # bind the leftmost button click to the draw\_circle method

        self.canvas.bind("<Button-1>", self.draw\_circle)

        # bind the leftmost button double click to the draw\_square method

        self.canvas.bind("<Double-Button-1>", self.draw\_square)

    def draw\_circle(self, event):

        """ Draw a magenta circle centered at the click position"""

        self.canvas.create\_oval(event.x - 10,

                                event.y - 10,

                                event.x + 10,

                                event.y + 10,

                                fill = "magenta")

    def draw\_square(self, event):

        """ Draw a cyan square at the click position"""

        self.canvas.create\_rectangle(event.x - 10,

                                event.y - 10,

                                event.x + 10,

                                event.y + 10,

                                fill = "cyan")

def main():

    # create the GUI application main window

    root = tkinter.Tk()

    # instantiate our drawing app object

    my\_app = DrawApp(root)

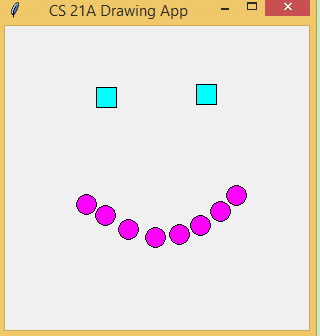
    # enter the main event loop and wait for events

    root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

 And here's our drawing app. Clicking or double clicking on the window results in a circle or a square drawn at the click position:



Let's examine our binding statements and compare it to our bind syntax:

widget.bind(event, handler)

self.canvas.bind("<Button-1>",self.draw\_circle)

Here self.canvas is our widget, "<Button-1>" is our event and self.draw\_circle is our our handler.

self.canvas.bind("<Double-Button-1>",self.draw\_square)

Here self.canvas is our widget, "<Double-Button-1>" is our event and self.draw\_square is our our handler.

Note that we are binding different events to different handlers **on the same widget.**

Now let's examine one of our handlers:

    def draw\_circle(self, **event**):

        """ Draw a magenta circle centered at the click position"""

        self.canvas.create\_oval(**event.x** - 10,

**event.y** - 10,

**event.x** + 10,

**event.y** + 10,

                                fill = "magenta")

We note that the event object is automatically passed to the handler.  **The handler has access to the coordinates of the event via the x and y attributes of the event object.** WINDOW coordinate

Assignment 10 questions:

1怎样check 一个rectangle 没被occupy? 没被涂色?

2 play method 怎样去access 已经create 的 rectangular 的color

或者 怎样改动其fill color?

老师 的 my\_label.config(text=”new text”)

**21. Assertions and Unit Testing**

**21.1. Assertions**

An assertion is a sanity-check that we can turn on when we are testing our program then turn off when we are done testing.

Assertions are a convenient way to insert **debugging** statements into a program.

Assertions are carried out by the assert statement.

Syntax:

assert expression

Or:

assert expression, description

If the assertion fails, Python raises **an AssertionError.** AssertionError exceptions can be caught and handled like any other exception using the try-except statement. If an assertion error is not handled, it will terminate the program and produce a traceback.

Let's try a few assertions:

>>> assert 1 + 1 == 2

The expression 1 + 1 == 2 is True, so the assert statement above does nothing.

>>> assert 1 + 1 > 8

Traceback (most recent call last):

AssertionError

1 + 1 > 8 is False, so the assert statement above raises an AssertionError.

We can also include a description that will only appear if the assertion fails.

>>> assert 1 + 1 > 0, "this is the message that will appear if the assertion fails"

>>> assert 1 + 1 > 5, "this is the message that will appear if the assertion fails"

Traceback (most recent call last):

AssertionError: this is the message that will appear if the assertion fails

**Optimize Flags:**

We can use the -O or -OO options on the Python command to reduce the size of a compiled module. **The -O flag removes assert statements, the OO flag removes both assert statements and docstrings.**

Both flags also set the special builtin name **\_\_debug\_\_** to False.

A program does NOT run faster when it is optimized. The only thing that is faster is the speed with which it is loaded.

Let's take a look at an example: the program below includes an assertion that will always be False.

#-------------------------------------------------------------------------------

# Name:        asserttest.py

# Purpose:     test assert statements with the -O flag

#

# Author:      Rula Khayrallah

#-------------------------------------------------------------------------------

"""

Include an assertion that is always False

"""

def main():

    assert 0 > 1, 'this assertion is always False'

    print('Hello World!')

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Now let's run the program without optimizing it from the terminal window:

>python asserttest.py

Traceback (most recent call last):

  File "asserttest.py", line 16, in <module>

    main()

  File "asserttest.py", line 12, in main

    assert 0 > 1, 'this assertion is always False'

AssertionError: this assertion is always False

Running the same program with the optimize flag:

>python **-O** asserttest.py

Hello World!

**Assertions or Exceptions?**

A good guideline is to **use an assertion to check for things that would only happen if there is a bug in our code**.

An **exception**, on the other hand is **used when we expect an error to happen** and we need to handle it.

Consider the following function that is supposed to compute a grade for a course based on the weights of several components.

def calculate\_grade(homework, midterm, final):

    """ Compute the overall numeric grade for the course based on the weights"""

   #  we first assert that the input to our function is valid

    assert homework <= 100 and homework >=0, "invalid grade "

    assert midterm <= 100 and midterm1 >=0, "invalid grade "

    assert final <= 100 and final >=0, "invalid grade "

    grade = (

            final \* 0.5 +

            homework \* 0.7 +

            midterm \* 0.1 +

            )

   # we assert that our computation results in a valid grade

    assert grade <= 100 and grade >=0, 'Invalid grade computation'

    return grade

The assertion may fail here because there is a bug in my code: the weights for the various grades do not add up to 1 so I may end up with a grade > 100.

On the other hand errors in user input are expected to occur even if our code is bug-free. These errors are handled with exceptions not assertions.

**21.2. Unit Testing and TDD**

Unit testing is an important part of any software development strategy.

It is important to write unit tests early and to **keep them up to date as code and requirements change.**

**Test Driven Development (TDD) advocates 主张writing tests before writing the code:**

* 1. we write the test based on how we would like the code to behave,
  2. then we write the code that makes the test pass.
  3. This forces us to **really understand the requirements before we start writing the code.**

**21.3. The unittest Module**

The Python Standard Library includes the **module** **unittest** which is a unit testing framework for Python.

Unittest supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To create tests using the Python unit testing framework, we first have to import the module:

import unittest

To unit test methods or functions in a given module, we also have to be able to access them.

So in order to test the Car class methods for example, we need to import car.

So now we have:

import unittest

import car

Now we can start writing our **test** cases.

**21.4. Test Cases and Test Fixtures**

A test case is an individual unit of testing. A test case checks for specific responses to a particular set of inputs.

The unittest module provides a base class (parent class), TestCase, which may be used to create new test cases.

**Defining a test case involves defining a subclass of TestCase. 定义测试案例 即 定义一个TestCase的实体**

This is how we create a new test case to test the Car class. Our test case class TestCar is a subclass of unittest.TestCase.

class TestCar(unittest.TestCase):

    """

    Test case for the normal execution of Car methods

    """

Before we test a certain method, we may have to do some initial setup.

**A test fixture represents the preparation needed to perform one or more tests, and any associated cleanup actions.**

The unit test framework supports test fixtures with the setUp and tearDown methods.

When a **setUp** method is defined for a given test case, it is invoked **prior** to each test.

Likewise, if a t**earDown** method is defined for a given test case, it is invoked **after** each test.

Let's define our test fixture for the TestCar test case. We'll create two cars with different initial conditions.

**setup method is used to create car objects from the tested object – Car class object**

def setUp(self):

        """

        Create three cars for testing.

        The first car has a fuel efficiency of 20 miles/gallom, initial

        mileage of 1000 miles and 9 gallons of fuel in the tank.

        The second car has a fuel efficiency of 30 miles/gallon, initial

        mileage of 5000 miles and 2 gallons of fuel in the tank

        The third car has a fuel efficiency of 24 miles/gallon, initial

        mileage of 0 miles and an empty gas tank.

        """

**self.first\_car** = **car.**Car('Porshe', '911', 20, 1000, 9)

**self.second\_car**= **car.**Car('Honda', 'Civic', 30, 2000, 2)

**self.third\_car**= **car**.Car('Honda', 'Pilot', 24)

Note that we **save the three car objects as instance variables so that we can access them in the test methods.**

Note also that to instantiate our Car class, we have to use its fully qualified name: car.Car.

We will not define a tearDown method for our test case.

**21.5. Tests and Assertion Methods**

Within a test case, we define individual tests by **defining methods** whose **names start with 'test'**.

def test\_drive(self):

""" test the drive method in the Car class"""

def test\_tank\_empty(self):

""" test the tank\_empty method in the Car class"""

Let's fill in the methods above:

def test\_drive(self):

        """ test the drive method in the Car class"""

        # driving when there is enough gas in the tank

        self.first\_car.drive(100)

**self.assertEqual(self.first\_car.mileage, 1100)**

**self.assertEqual(self.first\_car.gas\_in\_tank, 4)**

        # driving when there is not enough gas in the tank

        self.second\_car.drive(200)

**self.assertEqual(self.second\_car.gas\_in\_tank, 0)**

**self.assertEqual(self.second\_car.mileage, 2060)**

        # driving when there is no gas in the tank

        self.third\_car.drive(20)

**self.assertEqual(self.third\_car.gas\_in\_tank, 0)**

**self.assertEqual(self.third\_car.mileage, 0)**

This is our first test: we are testing the drive() method in the Car class.

We invoke the drive method on the three Car objects that we saved in out test case instance: self.first\_car, self.second\_car and self.third\_car. This was down in the setUp method.

Given our fixture we know what the expected mileage and gas in tank for each car should be after the drive method is invoked.

**assertEqual()** is a method defined on the **test case object**. **The call to assertEqual is used to check for the expected result.**

If the result is NOT equal to the one we provided, **an assertion exception is raised** but is **NOT reported on until the end of the tests**.

def test\_tank\_empty(self):

        """ test the tank\_empty method in the Car class"""

**self.assertFalse**(self.first\_car.tank\_empty())

**self.assertTrue**(self.third\_car.tank\_empty())

This is another test: we are testing the tank\_empty() method in the Car class.

**assertFalse()** and **assertTrue** are also methods defined on the test case object. Remember that self here refers to the test case object.

The TestCase class provides a number of assertion methods to check for and report failures.

These methods are used instead of the assert statement so that all the test results **can be accumulated and reported on at the end.**

Here's a list of the assertion methods available:

assertEqual

assertNotEqual

assertTrue

assertFalse

assertIs

assertIsNot

assertIsNone

assertIsNotNone

assertIn

assertNotIn

assertIsInstance

assertNotIsInstance

21.6. Unit Testing Example

This is how all the components fit together in the module testcar.py:

#-------------------------------------------------------------------------------

# Name:        testcar.py

# Purpose:     demonstrate the use of the unittest module

# Author:      Rula Khayrallah

#-------------------------------------------------------------------------------

import unittest

import car

class TestCar(unittest.TestCase):

    """

    Test case for the normal execution of Car methods

    """

    def setUp(self):

        """

        Create three cars for testing.

        The first car has a fuel efficiency of 20 miles/gallom, initial

        mileage of 1000 miles and 9 gallons of fuel in the tank.

        The second car has a fuel efficiency of 30 miles/gallon, initial

        mileage of 5000 miles and 2 gallons of fuel in the tank

        The third car has a fuel efficiency of 24 miles/gallon, initial

        mileage of 0 miles and an empty gas tank.

        """

        self.first\_car = car.Car('Porshe', '911', 20, 1000, 9)

        self.second\_car = car.Car('Honda', 'Civic', 30, 2000, 2)

        self.third\_car = car.Car('Honda', 'Pilot', 24)

    def test\_drive(self):

        """ test the drive method in the Car class"""

        # driving when there is enough gas in the tank

        self.first\_car.drive(100)

        self.assertEqual(self.first\_car.mileage, 1100)

        self.assertEqual(self.first\_car.gas\_in\_tank, 4)

        # driving when there is not enough gas in the tank

        self.second\_car.drive(200)

        self.assertEqual(self.second\_car.gas\_in\_tank, 0)

        self.assertEqual(self.second\_car.mileage, 2060)

        # driving when there is no gas in the tank

        self.third\_car.drive(20)

        self.assertEqual(self.third\_car.gas\_in\_tank, 0)

        self.assertEqual(self.third\_car.mileage, 0)

    def test\_tank\_empty(self):

        """ test the tank\_empty method in the Car class"""

        self.assertFalse(self.first\_car.tank\_empty())

        self.assertTrue(self.third\_car.tank\_empty())

**And here's our car module from section 10.8:**

class Car(object):

    """

    Represent a car in  my virtual world.

    Arguments:

    make (string): car make.

    model (string): car model.

    fuel\_efficiency (float): in miles per gallon.

    mileage (float, optional): current mileage on car in miles, defaults to 0.

    gas (float, optional): current gas in the tank in gallons, defaults to 0.

    Attributes:

    make (string): car make.

    model (string): car model.

    fuel\_efficiency (float): in miles per gallon.

    mileage (float): current mileage on the car in miles.

    gas\_in\_tank (float): current gas in the tank in gallons.

    """

    def \_\_init\_\_(self,  make,  model, fuel\_efficiency, mileage = 0, gas = 0):

        self.make = make

        self.model = model

        self.fuel\_efficiency = fuel\_efficiency

        self.mileage = mileage

        self.gas\_in\_tank = gas

    def tank\_empty(self):

        """Return True if the gas tank is empty."""

        return  self.gas\_in\_tank == 0

    def add\_gas(self, amount):

        """

        Add gas to the car.

        Parameter:

        amount (float): the amount of gas to be added in gallons.

        Returns:

        the updated car object (Car).

        """

        self.gas\_in\_tank += amount

        return self

    def drive(self, distance):

        """

        Drive a car a given distance, if possible.

        If there is not enough gas, drive as much as possible.

        Parameter:

        distance (float):  the distance to be driven in miles.

        Returns:

        the updated car object (Car)

        """

        max\_distance = self.fuel\_efficiency \* self.gas\_in\_tank

        if distance <= max\_distance:  # we can drive the distance

            self.mileage += distance

            self.gas\_in\_tank = self.gas\_in\_tank - distance /self.fuel\_efficiency

        else:   # not enough gas

            self.mileage += max\_distance  # drive as far as possible

            self.gas\_in\_tank = 0

        return self

**21.7. Running the Tests**

The test cases contained in the module testcar.py may be run from **the command line** (or terminal window) by **invoking the unittest module and passing testcar.py as its argument:**

>python -m unittest testcar.py

..

----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK

The two dots above indicate that two tests were run and passed.

We can also invoke unittest in **discovery mode:** this will run all the test cases found in modules whose names match a certain pattern, the default being test\*.py:

python -m unittest

\*\*\*Python modes:

-m mod : run library module as a script (terminates option list)

-O : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x

-OO : remove doc-strings in addition to the -O optimizations

\*\*\*

Now suppose we introduce an error in our drive method so that the mileage is not updated when there is not enough gas.

def drive(self, distance):

        """

        Drive a car a given distance, if possible.

        If there is not enough gas, drive as much as possible.

        Parameter:

        distance (float):  the distance to be driven in miles.

        Returns:

        the updated car object

        """

        max\_distance = self.fuel\_efficiency \* self.gas\_in\_tank

        if distance <= max\_distance:  # we can drive the distance

            self.mileage += distance

            self.gas\_in\_tank = self.gas\_in\_tank - distance /self.fuel\_efficiency

        else:   # not enough gas

**# comment the line below to introduce the error**

**# self.mileage += max\_distance  # drive as far as possible**

            self.gas\_in\_tank = 0

        return self

When we run the tests now, we get a different output.

>python -m unittest testcar.py

F.

======================================================================

FAIL: test\_drive (testcar.TestCar)

test the drive method in the Car class

----------------------------------------------------------------------

Traceback (most recent call last):

  File "C:\Users\Rula\Documents\CS21A\testcar.py", line 37, in test\_drive

    self.assertEqual(self.second\_car.mileage, 2060)

AssertionError: 2000 != 2060

----------------------------------------------------------------------

Ran 2 tests in 0.003s

FAILED (failures=1)

Note the F. above indicating that the first test failed and the second passed.

**22. More on Functions**

**22.1. Variable Number of Arguments**

We have already seen how to call a function or method using default arguments.

A **default argument** is an argument that assumes a default value if a value is not provided in the function call. That default value is specified in the function or method definition. The only restriction is that **non-default parameters appear before default parameters** in the function definition.

**def diff(a, b = 0):**

**return a - b**

When we provide a default value for b, we can call the function with two arguments:

>>> diff (4, 2)

2

Or with just one argument:

>>> diff (4 )

4

Note that the same rules apply to method arguments and class instantiations:

class Car(object):

    def \_\_init\_\_(**self,  make,  model, fuel\_efficiency, mileage = 0, gas = 0**):

>>> from car import Car

We can instantiate a Car object by specifying 5 arguments:

>>> my\_car = Car('Honda', 'Civic', 25, 10000, 12 )

Or just 4:

>>> my\_car = Car('Honda', 'Civic', 25, 10000)

Or just 3 arguments:

>>> my\_car = Car('Honda', 'Civic', 25 )

The unspecified arguments are then assigned the default value.

Sometimes we don't know beforehand how many arguments will passed to the function by the user and we don't have any default values.

We have the option to **specify that a function be called with an** **arbitrary 任意 number of arguments**. These arguments will be wrapped up in a **tuple**. **Before these variable arguments, zero or more normal arguments may occur. Normal parameter, (arbitrary 任意 argument1, arbitrary argument2,…)**

These arguments will be last in the list of formal parameters, because they scoop up (舀上来)all remaining input arguments that are passed to the function.

To specify a variable number of argument, we use **\*args** in the function definition.

We can also use **\*any\_other\_name** instead of \*args.

Here's an example:

def test\_args(req1, req2, **\*args**):

    """**this function takes at least two arguments**"""

    print('first required argument', req1)

    print('second required argument', req2)

**for arg in args:**

**print ('another argument', arg)**

>>> test\_args ('first', 'second', 'third', 'fourth')

first required argument first

second required argument second

another argument third

another argument fourth

>>> test\_args ('first')

Traceback (most recent call last):

TypeError: test\_args() missing 1 required positional argument: 'req2'

And here's a more useful example:

**def product(\*args):**

**"""**

**return the product of the arguments provided or 1 if no argument is provided**

**""""**

**result = 1**

**for number in args:**

**result = result \* number**

**return result**

>>> product(2, 3, 4, 5)

120

>>> product(5)

5

>>> product (2, 3)

6

>>> product()

1

**Variable Keyword Arguments**

We can also define a function or a method so that it accepts an arbitrary number of **keyword arguments**. **The keywords do not have to be defined beforehand**.

In the function or method definition we specify a final formal parameter of the form **\*\*kwargs**. We can also use \*\*any\_other\_name instead of \*\*kwargs.

The function then **receives a dictionary containing all keyword arguments** except those corresponding to a formal parameter or \*args.

**\*args**, if present, **must occur before \*\*kwargs.**

\*\*kwargs Example

**def custom\_print(name, \*\*kwargs):**

**"""this function takes at least one argument"""**

**print (name)**

**for key in kwargs:**

**print (key + ': '+ kwargs[key])**

>>> custom\_print('Alice', major='CS', email = 'alice@foothill.edu')

Alice

major: CS

email: alice@foothill.edu

>>> custom\_print('Bob', phone='555-3743')

Bob

phone: 555-3743

>>> custom\_print('Charlie')

Charlie

>>> custom\_print()

raceback (most recent call last):

TypeError: custom\_print() missing 1 required positional argument: 'name'

**22.2. Recursion**

We use recursion when we have a large problem that can be repeatedly broken down into one or more sub-problems.

To illustrate the process, we'll go back to our factorial example from module 4 and implement it using a recursive approach.

The factorial of a non-negative integer (!) is the product of all the integers from 1 up to and including the given integer.

By convention, the factorial of 0 is 1.

0! = 1

1! = 1 = 1

2! = 2 \* 1 = 2

3! = 3 \* 2 \* 1 = 6

4! = 4 \* 3 \* 2 \* 1 = 24

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

If we look closely at the above, we note that:

5! = 5 \* 4!

4!  = 4 \* 3!

3! = 3 \* 2!

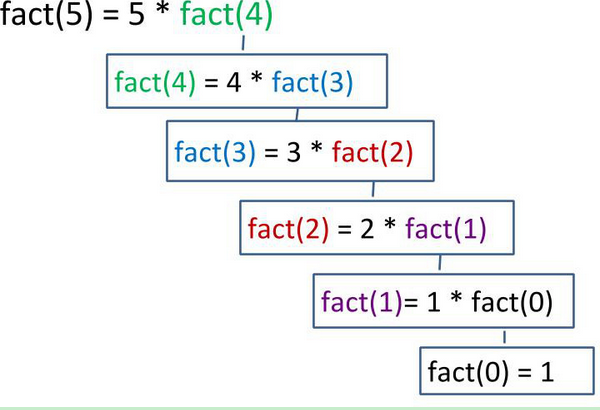
2! = 2 \* 1!

1! = 1 \* 0!

0! = 1

Now let's write a Python function fact that will compute the factorial of a given number. We'll call our function fact.

We know that:



In general for a given positive integer n:

fact(n) = n \* fact(n - 1)

This is known as the **recursive rule:** it is the rule by which **we express the problem in terms of one or more smaller sub-problems**. The smaller sub-problem here is computing the factorial of n - 1. To solve the smaller sub-problem, **the function calls itself with a smaller argument**.

Eventually the function will come across **a sub-problem that is small enough so that it can handle it without calling itself.** This is known as **the base case**. Our base case here is fact(0). We know that: **fact(0) = 1.**

**In the base case, the problem-size is as small as it can be.**

The base case is needed to prevent the function from calling itself over and over again without stopping: **the base case stops the recursion.**

In the base case, **the function knows the result or can compute it using some straightforward formula.**

Now we are ready to write the following recursive definition of our factorial function:

def fact(number):

    """ compute the factorial of a non-negative integer """

    # base case: the problem-size is as small as it can be

    if number == 0:

        return 1;

    else:

        # recursive rule: the function calls itself with a smaller argument

        return number \* **fact(number - 1)**

**The general structure of a recursive function** is as follows:

def recursive\_function(**problem**):

    if **base case condition:**

        return **base case result**

    else: # recursive rule defined as return content

        return some result that involves **recursive\_function(sub\_problem)**

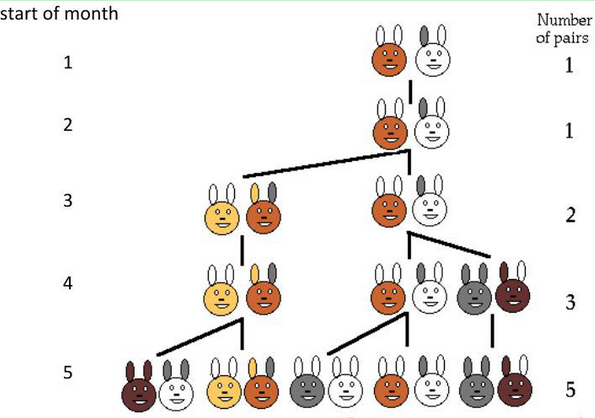
**22.3. Fibonacci Numbers Example**

To illustrate a more complex recursive example, we'll examine Fibonacci numbers. Fibonacci numbers appear in several mathematical areas. We'll take a look at one illustration that describes the growth of a rabbit population.

First we assume that a newly born pair of rabbits are put in a field. At the end of its second month a pair of rabbits produces another pair of rabbits. Rabbits never die and each pair always produces one new pair every month from the second month on.

Our task is to determine how many pairs of rabbits will be in the field in one year.

Let fibonacci(n) denote the number of pairs of rabbits at the start of month n.



fibonacci(0) is 0.

fibonacci(1) is 1 since there is one pair of rabbits at the start of month 1.

Then from month 2 on, the rabbits in the field at the start of month n are:

the same rabbits in the field at the start of month n - 1 (since rabbits never die)

+ the newborn rabbits.

Since each each pair of rabbits produces one new pair every month **from their second month** on, the number of newborn rabbits at the start of month n is equal to the number of rabbits that are at least 2 months old.

So the number of newborn pairs at the start of month n is equal to the number of pairs of rabbits in the field two months earlier: fibonacci(n - 2).

So we can write:

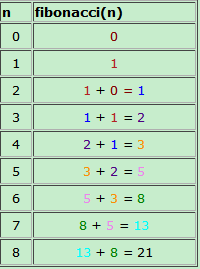
fibonacci(n) = fibonacci(n - 1) + fibonacci(n-2)

This will be our recursive rule. Note that in this case, we are formulating the problem in terms of **two smaller sub-problems.**

Our base case:

fibonacci(0) = 0

fibonacci(1) = 1



We can now write our recursive fibonacci function as follows:

def fibonacci(n):

    """

    return the Fibonacci number Fn

    """

    # base case: 0 or 1

    if n <= 1:

        return n

    else:  # recursive rule

        return **fibonacci(n-1) + fibonacci(n-2)**

We still need to answer the question: how many pairs of rabbits will there be in the field in one year?

**In one year**, **at the beginning of month 13**, there will be: fibobacci(13) pairs of rabbit:

def main():

    print('In one year, there will be', fibonacci(13), 'pairs of rabbits.')

We get:

In one year, there will be 233 pairs of rabbits.

The recursive implementation above is natural and elegant but not efficient.

The fibonacci function is called 15 times to compute fibonacci(5). It is called 753 times to compute fibonacci(13).

In Python, the alternate solution using a list and a for loop is much more efficient, but not as elegant:

def fibonacci(n):

    """

    return the Fibonacci number Fn - non recursive solution

    """

    fib = [0,1] # keep a list of the last 2 known fibonacci numbers

    for i in range(2, n+1):

       # calculate fibonacci(i) and append it to the list

        fib.append(fib[-1]+fib[-2])

        del fib[0]   # we only need the last two numbers in the sequence

return fib[-1]  # the last item in the list is fibonacci(n)

**23.1. The OS Module**

The OS module is another module included in the Python standard library.

It offers a platform independent way to access and manipulate local directories, files and environment variables. It has a **consistent interface across supported operating systems** so our programs can run on any platform.

We have already seen how to use the **os.getcwd()** function to figure out what our working directory is.

On Windows, from the PyCharm console, we can type the following:

>>> import os

>>> os.getcwd()

'C:**\\**Users**\\**Rula**\\**Documents**\\**CS21A'

We get the backslashes here because this example is on Windows. One backslash shows as a double backslash because of the escaping issue with strings.

On a Mac, we get the following:

>>> import os

>>> os.getcwd()

'/Users/rulakhayrallah/PycharmProjects/CS21A'

When we specify a filename with a relative path, that relative path will start from our working directory. The **os.chdir()** function lets us change the current working directory.

On Windows:

>>> import os

>>> os.getcwd()

'C:\\Users\\Rula\\Documents\\CS21A'

**>>> os.chdir('/users')**

>>> os.getcwd()

'C:\\users'

>>> **os.chdir('Rula')**

>>> os.getcwd()

'C:\\users\\Rula'

>>> **os.chdir('Documents/CS21A')**

>>> os.getcwd()

'C:\\users\\Rula\\Documents\\CS21A'

With all the os functions, **we can use forward slashes across all the platforms (even on Windows.)**

If we run this example on Mac OS X, we get the following.

>>> import os

>>> os.getcwd()

'/Users/rulakhayrallah/PycharmProjects/CS21A'

>>> os.chdir('/users')

>>> os.getcwd()

'/Users'

>>> os.chdir('rulakhayrallah')

>>> os.getcwd()

'/Users/rulakhayrallah'

>>> os.chdir('PycharmProjects/CS21A')

>>> os.getcwd()

'/Users/rulakhayrallah/PycharmProjects/CS21A'

**os.sep** will show us the character used to separate pathname components on our system.

On Windows:

>>> os.sep

'\\'

On a Mac:

>>> os.sep

'/'

getcwd(), chdir() as well as sep are all in the os namespace. We type os.getcwd, os.chdir and os.sep to access them.

Next we’ll see a number of functions that are in the os.path namespace. **os.path is a submodule of os**. We don’t need to import it separately if we have already imported os.

The os.path submodule includes the following functions: os.path.join(), os.path.split(), os.path.splitext(), os.path.realpath(), os.path.exists() and os.path.isfile().

**os.path.join()**

The os.path.join() function intelligently constructs a pathname out of one or more partial pathnames.

If any component is an absolute path, all previous components are thrown away, and joining continues from there on.

Join adds exactly one separator following each non-empty part except the last.

Here's an example on Windows:

>>> os.path.join('users/rula/documents', 'cs21a', 'tests')

'users/rula/documents\\cs21a\\tests'

If any component is an absolute path, all previous components are thrown away:

>>> os.path.join('users/rula/documents', '**c://users/tests**')

'c://users/tests'

>>> os.path.join('users/rula/documents', 'cs21a', 'tests', 'crawled.txt')

'users/rula/documents\\cs21a\\tests\\crawled.txt'

Here's a similar example on a Mac:

>>> os.path.join('users/rulakhayrallah', 'cs21a', 'tests')

'users/rulakhayrallah/cs21a/tests'

If any component is an absolute path, all previous components are thrown away:

>>> os.path.join('users/rulakhayrallah', **'/users/guest/test'**)

'/users/guest/test'

>>> os.path.join('users/rulakhayrallah/documents', 'cs21a', 'tests', 'crawled.txt')

'users/rulakhayrallah/documents/cs21a/tests/crawled.txt'

**os.path.split():**

The os.path.split() function splits a full pathname and returns a tuple containing the path and filename. We can then use tuple unpacking to get each of the separate variables.

>>> os.path.split('/users/rula/Documents/CS21A/hello.py')

('/users/rula/Documents/CS21A', 'hello.py')

We can also write, using tuple unpacking:

>>> (pathname, filename) = os.path.split('/users/rula/Documents/CS21A/hello.py')

>>> pathname

'/users/rula/Documents/CS21A'

>>> filename

'hello.py'

Note that the file name may be empty:

>>> import os

>>> pathname, filename = os.path.split('users/Rula/')

>>> pathname

'users/Rula'

>>> filename

' '

However, if we omit the / at the end:

>>> pathname, filename = os.path.split('users/Rula')

>>> pathname

'users'

>>> filename

'Rula'

**os.path.splitext():**

The os.path.splitext() function (ext for extension not text) splits a filename and returns a tuple containing the filename and the file extension. We can then use tuple unpacking to get each of the separate variables.

>>> filename, ext = os.path.splitext ('users/Rula/CS21A/hello.py')

>>> filename

'users/Rula/CS21A/hello'

>>> ext

'.py'

>>> filename, ext = os.path.splitext ('hello.py')

>>> filename

'hello'

>>> ext

'.py'

The period separator is included in the extension here

Here again, the extension may be empty.

>>> filename, ext = os.path.splitext ('hello')

>>> filename

'hello'

>>> ext

''

**os.path.realpath():**

The os.path.realpath() function returns the absolute path of the specified filename, eliminating any symbolic links encountered in the path.

Here's an example on Windows:

>>> os.getcwd()

'C:\\users\\Rula\\Documents\\CS21A'

>>> os.path.realpath('hello.py')

'C:\\users\\Rula\\Documents\\CS21A\\hello.py'

>>> os.path.realpath('/test/hello.py')

'C:\\test\\hello.py'

>>> os.path.realpath('test/hello.py')

'C:\\users\\Rula\\Documents\\CS21A\\test\\hello.py'

>>> os.path.realpath('../hello.py')

'C:\\users\\Rula\\Documents\\hello.py'

>>> os.path.realpath('../../hello.py')

'C:\\users\\Rula\\hello.py'

>>> os.path.realpath('./test/hello.py')

'C:\\users\\Rula\\Documents\\CS21A\\test\\hello.py'

And a similar example **on a Mac**:

>>> os.getcwd()

'/Users/rulakhayrallah/PycharmProjects/CS21A'

>>> os.path.realpath('hello.py')

'/Users/rulakhayrallah/PycharmProjects/CS21A/hello.py'

>>> os.path.realpath('/test/hello.py')

'/test/hello.py'

>>> os.path.realpath('test/hello.py')

'/Users/rulakhayrallah/PycharmProjects/CS21A/test/hello.py'

>>> os.path.realpath('../hello.py')

'/Users/rulakhayrallah/PycharmProjects/hello.py'

>>> os.path.realpath('../../hello.py')

'/Users/rulakhayrallah/hello.py'

>>> os.path.realpath('./test/hello.py')

'/Users/rulakhayrallah/PycharmProjects/CS21A/test/hello.py'

**No Guarantee of Existence**

Note that the path functions that we have seen so far (join, split, splitext and realpath) can be used to parse strings representing filenames into their component parts and to assemble components into absolute paths.

It is important to realize that **these functions do not depend on the paths actually existing; they operate solely on the strings.**

What if we need to verify that a path or a file exists?

**os.path.exists(path)** returns True if the path exists. It works for both files and directories.

**os.path.isfile(path**) returns True if the specified **path**  is an existing file. It returns False if the path is a directory.

>>> os.path.exists('/users/Rula')

True

>>> os.path.isfile('/users/Rula')

False

>>> os.path.exists('hello.py')

True

>>> os.path.isfile('hello.py')

True

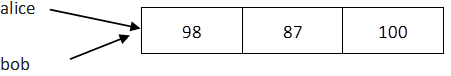
>>> os.path.isfile('nofile.py')

False

**23.2. Copy and Deepcopy**

Remember our discussion about variables '**referring**' to objects as opposed to 'containing' them?

We have seen that **an assignment statement on lists does not make a copy**. Instead, the assignment makes the two variables point to the same list in memory.



>>> alice=[98,87,100]

>>> bob=alice

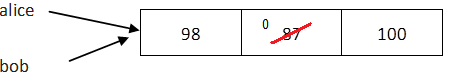
>>> alice

[98, 87, 100]

>>> bob

[98, 87, 100]

What happens if we now change one list item in alice?

>>> alice[1]=0 

>>> alice

[98, 0, 100]

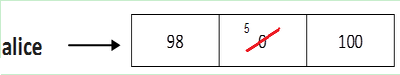
>>> bob

[98, 0, 100]

What if we wanted bob to have **a different copy of the list**, one that initially has the same values as alice but that is not affected by future changes to alice.

If the list we are copying **is not nested**, the following slice assignment will work.

>>> alice=[98, 0, 100]





>>>bob=alice[:]

>>> bob

[98, 0, 100]

>>> alice

[98, 0, 100]

>>> alice[1]=5

>>> alice

[98, 5, 100]

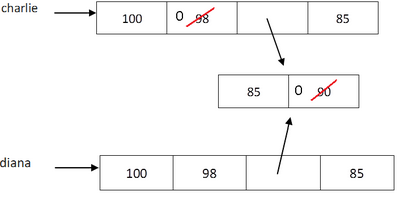
>>> bob

[98, 0, 100]

However with nested lists the slice assignment will NOT work. **It is a shallow copy**.

>>> charlie = [100,98,[85,90],85]

>>> diana=charlie[:]



>>> diana

[100, 98, [85, 90], 85]

>>> charlie[1]=0

>>> charlie

[100, 0, [85, 90], 85]

>>> diana

[100, 98, [85, 90], 85]

>>> charlie[2][1]=0

>>> charlie

[100, 0, [85, 0], 85]

>>> diana

[100, 98, [85, 0], 85]

**The copy module:**

The shallow versus deep copy issue is relevant not just for lists but for compound objects in general.

The copy module offers an alternative to deal with it.  It has 2 distinct functions:

copy.copy()

copy.deepcopy()

The copy.copy() function  constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

**For lists, copy.copy() is equivalent to the slice copy.**

The copy.deepcopy() function constructs a new compound object and then, **recursively, inserts copies into it of the objects found in the original.**

**A deep copy duplicates the compound object as well as the contents of all of its contained objects.**

Here's a copy and deepcopy list example:

>>>import copy

>>> charlie = [100,98,[85,90],85]

>>> diana=copy.copy(charlie)

>>> frank=copy.deepcopy(charlie)

>>> charlie[1]=0

>>> charlie

[100, 0, [85, 90], 85]

>>> diana

[100, 98, [85, 90], 85]

>>> frank

[100, 98, [85, 90], 85]

>>> charlie[2][1]=0

>>> charlie

[100, 0, [85, 0], 85]

>>> diana

[100, 98, [85, 0], 85]

>>> frank

[100, 98, [85, 90], 85]

And here's a compound object example:

class Profile(object):

    """

    Represent a person's social profile

    Argument:

    name (string): a person's name - assumed to uniquely identify a person

    Attributes:

    name (string): a person's name - assumed to uniquely identify a person

    statuses (list): a list containing a person's statuses

    friends (set): set of friends for the given person.

                   it is the set of profile objects representing these friends.

    """

    def \_\_init\_\_(self,  name):

        self.name = name

        self.statuses = []

        self.friends = set()

>>> from social import Profile

>>> import copy

>>> alice = Profile('Alice')

>>> bob = Profile('Bob')

>>> charlie = Profile('Charlie')

>>> alice.add\_friend(bob)

>>> copy\_alice = copy.copy(alice)

>>> deepcopy\_alice = copy.deepcopy(alice)

>>> alice.get\_friends()

['Bob']

>>> copy\_alice.get\_friends()

['Bob']

>>> deepcopy\_alice.get\_friends()

['Bob']

>>> alice.add\_friend(charlie)

>>> alice.get\_friends()

['Bob', 'Charlie']

>>> copy\_alice.get\_friends()

['Bob', 'Charlie']

>>> deep\_copy.get\_friends()

['Bob']

**Deepcopy and Object Identity:**

>>> copy\_alice is alice

False

>>> deepcopy\_alice is alice

False

>>> copy\_alice.friends is alice.friends

True

>>> deepcopy\_alice.friends is alice.friends

False

CHAT app Server , Client side

**Sys.stdout.flush():**

Python's standard out is buffered (meaning that it collects some of the data "written" to standard out before it writes it to the terminal). Calling sys.stdout.flush() forces it to "flush" the buffer, meaning that it will write everything in the buffer to the terminal, even if normally it would wait before doing so.

Python Network

partial reads/writes—partial data transfer

• Be aware that for TCP, the data stream is

continuous---no concept of records, etc.

30

# Client

...

s.send(data)

s.send(moredata)

...

# Server

...

data = s.recv(maxsize)

...

This recv() may return data

from both of the sends

combined or less data than

even the first send

• A lot depends on OS buffers, network

bandwidth, congestion, etc.

Here is a programming template for reassembling data

33

fragments = [] # List of chunks

while not done:

chunk = s.recv(maxsize) # Get a chunk

if not chunk:

break # EOF. No more data

fragments.append(chunk)

# Reassemble the message

message = "".join(fragments)