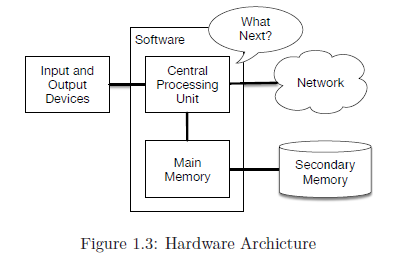
Python Application Programming

# MODULE 1

## Why Learn Programming Language

* Solve complex problems
* Operate instructions on machine
* Better handling of data
* Creativity, Motivation and Rewarding

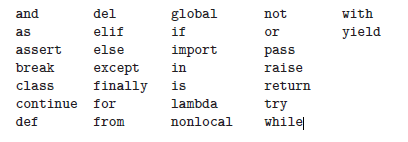
## Computer Hardware Architecture



* CPU
* Main Memory
* Secondary Memory
* IO Devices
* Network

## Words and Sentences

* Reserve Words



* E.g. for a sentence in python

print ('Hello world!')

## Conversing with Python

* Python is amazingly complex and powerful and very picky about the syntax

you use to communicate with it

* Python is not intelligent. You are really just having a conversation with

yourself, but using proper syntax.

## Interpreter and compiler

Interpreters

* An interpreter reads the source code of the program as written by the

programmer, parses the source code, and interprets the instructions on the

fly.

* Python is an interpreter and when we are running Python interactively, we

can type a line of Python (a sentence) and Python processes it immediately

and is ready for us to type another line of Python.

* Some of the lines of Python tell Python that you want it to remember some

value for later.

Compiler

* A compiler needs to be handed the entire program in a file, and then it runs a

process to translate the high-level source code into machine language and

then the compiler puts the resulting machine language into a file for later

execution.

## Writing a Program

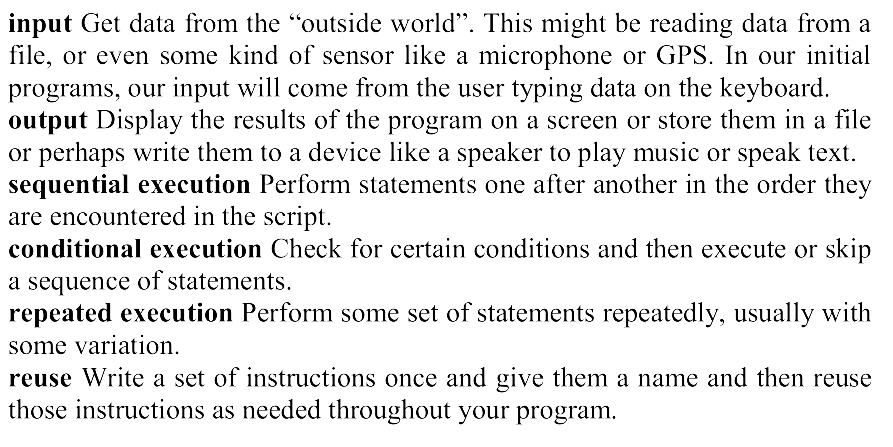
* When we want to write a program, we use a text editor to write the Python

instructions into a file, which is called a script.

* By convention, Python scripts have names that end with .py.
* To execute the script, you have to tell the Python interpreter the name of the

file.

## Building Blocks of a Program



## Errors

Syntax errors

* These are the first errors you will make and the easiest to fix.
* A syntax error means that you have violated the “grammar” rules of Python.
* Python does its best to point right at the line and character where it noticed it

was confused.

* The only tricky bit of syntax errors is that sometimes the mistake that needs

fixing is actually earlier in the program than where Python noticed it was

confused.

Logic errors

* A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another.

Semantic Error

* A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program.

## Values and Types

* A value is one of the basic things a program works with, like a letter or a number.
* Strings belong to the type str and integers belong to the type int.
* numbers with a decimal point belong to a type called float,

because these numbers are represented in a format called floating point.

* Do not use commas between the numbers

# Variables, Statements and Operators

* A variable is a name that refers to a value.
* An assignment statement creates new variables and gives them values
* Programmers generally choose names for their variables that are meaningful

and document what the variable is used for.

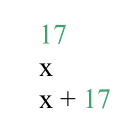
* Can have alphanumeric characters and \_ It should not start with number
* A statement is a unit of code that the Python interpreter can execute

Operators are special symbols that represent computations like addition and

multiplication.

* The values the operator is applied to are called operands.
* The operators +, -, \*, /, and \*\* perform addition, subtraction, multiplication,

division, and exponentiation. + is used for concatenation in strings

* An expression is a combination of values, variables, and operators
* A value all by itself is considered an expression
* 

## Order of Execution

* When more than one operator appears in an expression, the order of

evaluation depends on the rules of precedence.

* For mathematical operators, Python follows mathematical convention.
* The acronym PEMDAS is a useful way to remember the rules:
* Parentheses
* Exponentiation has the next highest precedence
* Multiplication and Division have the same precedence, which is higher

than Addition and Subtraction, which also have the same precedence.

* Operators with the same precedence are evaluated from left to right.

## Modulus Operator

* The modulus operator works on integers and yields the remainder when the first operand is divided by the second.
* In Python, the modulus operator is a percent sign (%)
* // operator is used to get quotient

## Input, Comments, Mnemonics

* Input () is used to take Input
* # is used for comments
* We must choose meaningful mnemonics

## Conditional Execution

* A Boolean expression is an expression that is either true or false. The

operator ==, which compares two operands and produces True if they are equal and False otherwise

* There are three logical operators: and, or, and not.
* In order to write useful programs, we almost always need the ability to

check conditions and change the behaviour of the program accordingly.

* Conditional statements give us this ability.



* Statements like if are called compound statements because they stretch

across more than one line.

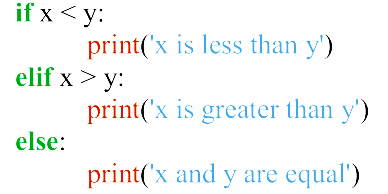
* A second form of the if statement is alternative execution, in which there are

two possibilities and the condition determines which one gets executed.

## 

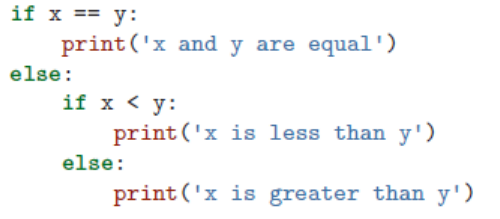
* The alternatives are called branches, because they are branches in the flow

of execution.



* One conditional can also be nested within another. We could have written

the three-branch example like this



## Try and Except blocks

* We use Try and Except blocks to handle exceptions

Here is a sample program to convert a Fahrenheit temperature to a Celsius

temperature:

inp = input ('Enter Fahrenheit Temperature: ')

fahr = float(inp)

cel = (fahr - 32.0) \* 5.0 / 9.0

print(cel)

* If we execute this code and give it invalid input, it simply fails with an

unfriendly error message:

python fahren.py

Enter Fahrenheit Temperature:72

22.22222222222222

python fahren.py

Enter Fahrenheit Temperature: Fred

Traceback (most recent call last):

File "fahren.py", line 2, in <module>

fahr = float(inp)

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

* We can rewrite our temperature converter as follows:

inp = input ('Enter Fahrenheit Temperature:')

try:

fahr = float(inp)

cel = (fahr - 32.0) \* 5.0 / 9.0

Source diginotes.in

print(cel)

except:

print ('Please enter a number')

## Short Circuit evaluation of expressions

* When Python is processing a logical expression such as x >= 2 and (x/y) > 2,

it evaluates the expression from left to right.

* Because of the definition of and, if x is less than 2, the expression x >= 2 is

False and so the whole expression is False regardless of whether (x/y) > 2

evaluates to True or False.

* When Python detects that there is nothing to be gained by evaluating the rest

of a logical expression, it stops its evaluation and does not do the

computations in the rest of the logical expression.

* When the evaluation of a logical expression stops because the overall value

is already known, it is called short-circuiting the evaluation.

* The short-circuit behaviour leads to a clever technique called the guardian

pattern.

# Functions

* a function is a named sequence of statements that performs a computation.
* When you define a function, you specify the name and the sequence of

Statements

* Later, you can “call” the function by name.
* The expression in parentheses is called the argument of the function.
* The argument is a value or variable that we are passing into the function as

input to the function.

* The result, for the type function, is the type of the argument.
* It is common to say that a function “takes” an argument and “returns” a

result.

* The result is called the return value.

## Built-in Functions

* Python provides a number of important built-in functions that we can use

without needing to provide the function definition.

* The max and min functions give us the largest and smallest values in a list,

respectively:

* >>> max ('Hello world')

'w'

>>> min ('Hello world')

* The max function tells us the “largest character” in the string
* Another very common built-in function is the len function which tells us how many items are in its argument.
* len ('Hello world')
* 11
* These functions are not limited to looking at strings. They can operate on

any set of values

* You should treat the names of built-in functions as reserved words
* Type conversion functions can convert one type to another. E.g.: int (), float (), str ()

## Random Numbers

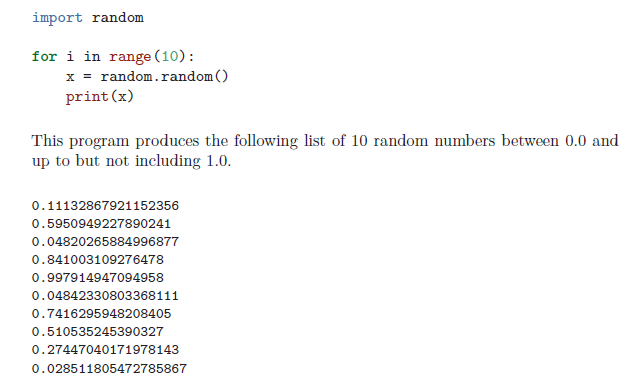
* Pseudorandom numbers are not truly random because they are generated by

a deterministic computation, but just by looking at the numbers it is all but

impossible to distinguish them from random.

* The random module provides functions that generate pseudorandom

Numbers

* The function random returns a random float between 0.0 and 1.0 (including0.0 but not 1.0).
* The function randint takes the parameters low and high, and returns an

integer between low and high (including both).

>>> random.randint(5, 10)

5

* To choose an element from a sequence at random, you can use choice:

>>> t = [1, 2, 3]

>>> random.choice(t)

2

>>> random.choice(t)

3

* The random module also provides functions to generate random values from

continuous distributions including Gaussian, exponential, gamma, and a few

more.

## Math Functions

* Python has a math module that provides most of the familiar mathematical

functions.

* We need to import before using
* The module object contains the functions and variables defined in the

module.

* To access one of the functions, you have to specify the name of the module

and the name of the function, separated by a dot (also known as a period).

This format is called dot notation.

|  |
| --- |
| * math.log10(ratio) [also loge()] |
| * math.sin(radians) [also other trigonometric functions] |
| * math.sqrt(2) |
| * math.pi accurate upto 15 digits |

## Adding new functions

* A function definition specifies the name of a new function and the sequence

of statements that execute when the function is called.

def print\_lyrics():

print("I'm a lumberjack, and I'm okay.")

print('I sleep all night and I work all day.')

* def is a keyword that indicates that this is a function definition.
* The name of the function is print\_lyrics.
* The rules for function names are the same as for variable names: letters,

numbers and some punctuation marks are legal, but the first character can’t

be a number.

* You can’t use a keyword as the name of a function, and you should avoid

having a variable and a function with the same name.

* The empty parentheses after the name indicate that this function doesn’t take

any arguments.

* Later we will build functions that take arguments as their inputs.
* The first line of the function definition is called the header; the rest is called

the body.

* The header has to end with a colon and the body has to be indented.
* By convention, the indentation is always four spaces.
* The body can contain any number of statements.
* in interactive mode, the interpreter prints

ellipses (. . . ) to let you know that the definition isn’t complete

* To end the function, you have to enter an empty line
* Defining a function creates a variable with the same name.
* >>> print(print\_lyrics)
* <function print\_lyrics at 0xb7e99e9c>
* >>> print(type(print\_lyrics))
* <class 'function'>
* The value of print\_lyrics is a function object, which has type “function”.
* The syntax for calling the new function is the same as for built-in functions:
* >>> print\_lyrics()
* I'm a lumberjack, and I'm okay.
* I sleep all night and I work all day.
* Once you have defined a function, you can use it inside another function.

## Flow of Execution

In order to ensure that a function is defined before its first use, you have to know

the order in which statements are executed, which is called the flow of execution.

Execution always begins at the first statement of the program. Statements are

executed one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but

remember

that statements inside the function are not executed until the function is

called.

A function call is like a detour in the flow of execution. Instead of going to the

next

statement, the flow jumps to the body of the function, executes all the statements

there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another.

While in the middle of one function, the program might have to execute the

statements

in another function. But while executing that new function, the program

might have to execute yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function

completes, the program picks up where it left off in the function that called it.

When it gets to the end of the program, it terminates.

## Arguments and Parameters

* Inside the function, the arguments are assigned to variables called parameter
* Here is an example of a user-defined function that takes an argument:
* def print\_twice(bruce):
* print(bruce)
* print(bruce)

## Fruitful and Void functions

* Some of the functions we are using, such as the math functions, yield results;

for lack of a better name, I call them fruitful functions.

* x = math.cos(radians)
* golden = (math.sqrt(5) + 1) / 2
* Other functions, like

print\_twice, perform an action but don’t return a value. They are called void

functions.

* print(x)

# MODULE-2

# Iteration

* A common pattern in assignment statements is an assignment statement that

updates a variable, where the new value of the variable depends on the old.

x = x + 1

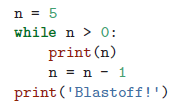
* If you try to update a variable that doesn’t exist, you get an error, because

Python evaluates the right side before it assigns a value to x

* Before you can update a variable, you have to initialize it
* Updating a variable by adding 1 is called an increment; subtracting 1 is

called a decrement.

## While Loop

* One form of iteration in Python is the while statement.
* 
* Flow of Execution: Evaluate the condition, yielding True or False.
* If the condition is false, exit the while statement and continue execution at

the next statement.

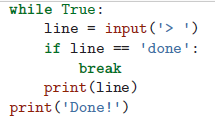
* If the condition is true, execute the body and then go back to step 1
* We call each time we execute the body of the loop an iteration.
* If there is no iteration variable, the loop will repeat forever, resulting in an

infinite loop.

## Infinite Loops and Break

* you can write an infinite loop on purpose and then use the break

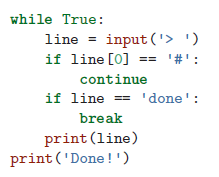
statement to jump out of the loop.

* 

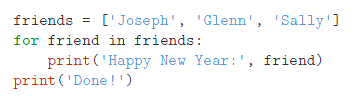
## Continue

* you can use the continue statement to skip to the next iteration

without finishing the body of the loop for the current iteration.

* 

## For

* Used to loop through a set of things such as a list of words, the lines in a file, or a list of numbers
* Definite loop can be constructed using a for statement
* 

## Looping patterns

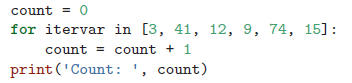
* Loops are generally constructed by:

• Initializing one or more variables before the loop starts

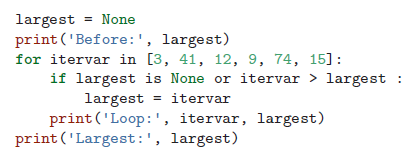
• Performing some computation on each item in the loop body,

possibly changing the variables in the body of the loop

• Looking at the resulting variables when the loop complete

* 

1 Counting

* 

# Strings

* A string is a sequence of characters.
* You can access the characters one at a time with the bracket operator
* 
* The second statement extracts the character at index position 1 from the fruit

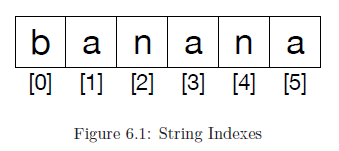
variable and assigns it to the letter variable.

* The expression in brackets is called an index.
* But in Python, the index is an offset from the beginning of the string, and the

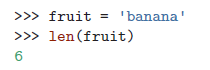
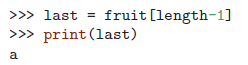
offset of the first letter is zero.

* You can use any expression, including variables and operators, as an index,

but the value of the index has to be an integer.

* 

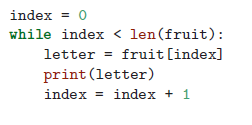
## Getting the length of the string

* len is a built-in function that returns the number of characters in a string
* 
* Since we started counting at zero, the six letters are numbered 0 to 5.
* To get the last character, you have to subtract 1 from length
* 
* Alternatively, you can use negative indices
* The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last.

## Traversing through a string using a Loop

* One way to write a traversal is with a while loop
* This loop traverses the string and displays each letter on a line by itself
* The last character accessed is the one with the index len(fruit)-1, which is

the last character in the string.

* 
* Another way to write a traversal is with a for loop
* 
* Each time through the loop, the next character in the string is assigned to the

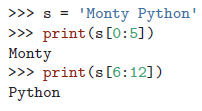
variable char.

* The loop continues until no characters are left.

## String slicing

* A segment of a string is called a slice. Selecting a slice is similar to selecting

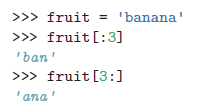
a character

* 
* The operator returns the part of the string from the “n-eth” character to the

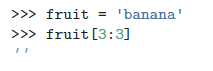
“m-eth” character, including the first but excluding the last.

* If you omit the first index (before the colon), the slice starts at the beginning

of the string. Similarly omitting index after colon will make the slice end at last index

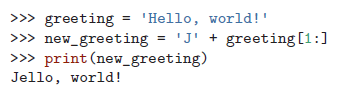
* 
* If the first index is greater than or equal to the second the result is an empty

string, represented by two quotation marks

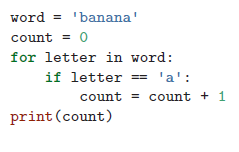
* 
* An empty string contains no characters and has length 0, but other than that,

it is the same as any other string.

## Strings are immutable

* You can’t change an existing string
* The best you can do is create a new string that is a variation on the original
* 
* This example concatenates a new first letter onto a slice of greeting.
* It has no effect on the original string

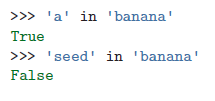
## Looping and counting in string

* 

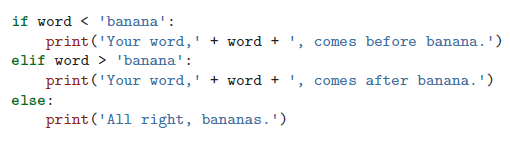
## The in operator

* The word in is a boolean operator that takes two strings and returns True if

the first appears as a substring in the second:

* 

## String comparison

* The comparison operators work on strings.
* 
* 
* All the uppercase letters come before all the lowercase letters
* 
* A common way to address this problem is to convert strings to a standard

format, such as all lowercase, before performing the comparison

## String Methods

* Strings are an example of Python objects.
* An object contains both data (the actual string itself) and methods, which are

effectively functions that are built into the object and are available to any

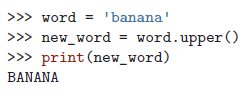
instance of the object

* Python has a function called dir which lists the methods for an object
* The type function shows the type of an object and the dir function shows the

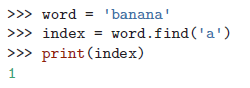
available methods.

* We call a method by appending the method name to the variable name using

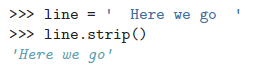
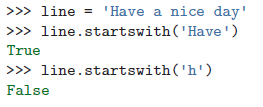
the period as a delimiter.

* 
* The empty parentheses indicate that this method takes no argument.
* A method call is called an invocation; in this case, we would say that we are

invoking upper on the word.

* there is a string method named find that searches for the position of one string within another
* 
* 
* It can take as a second argument the index where it should start:
* 
* One common task is to remove white space (spaces, tabs, or newlines) from

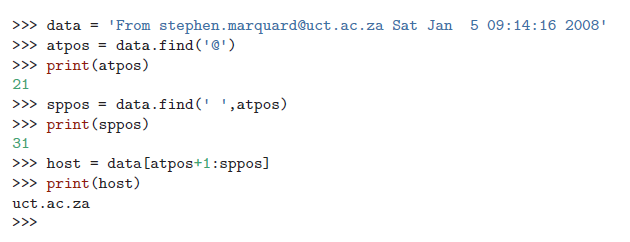
the beginning and end of a string using the strip method

* 
* Some methods such as startswith return boolean values
* 
* startswith requires case to match

## Parsing a String

* Often, we want to look into a string and find a substring.
* We use a version of the find method which allows us to specify a position in

the string where we want find to start looking.

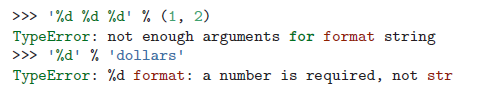
* 
* When we slice, we extract the characters from “one beyond the at-sign

through up to but not including the space character”.

## Format Operator %

* Allows us to replace parts of strings by data stored in variables
* %d,%f,%s etc
* 
* The number of elements in the tuple must match the number of format sequences

in the string. The types of the elements also must match the format sequences

* 

# Files

## Opening Files

* When we want to read or write a file (say on your hard drive), we first must

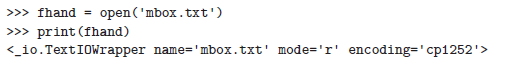
open the file.

* Opening the file communicates with your operating system, which knows

where the data for each file is stored.

* When you open a file, you are asking the operating system to find the file by

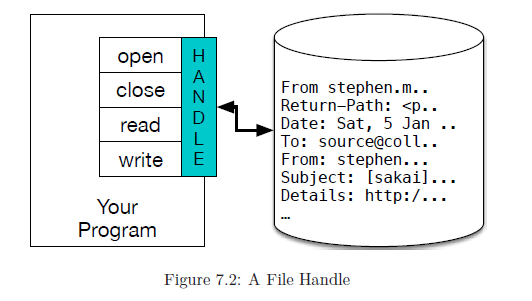
name and make sure the file exists.

* 
* If the open is successful, the operating system returns us a file handle.
* The file handle is not the actual data contained in the file, but instead it is a

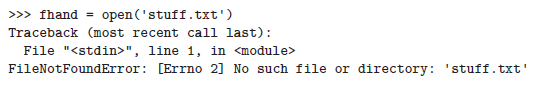
“handle” that we can use to read the data.

* You are given a handle if the requested file exists and you have the proper

permissions to read the file.

* 
* If the file does not exist, open will fail with a traceback and you will not get

a handle to access the contents of the file:

* 

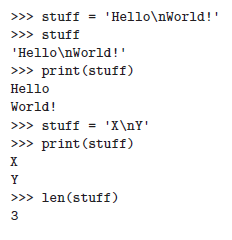
## Text Files and Lines

* A text file can be thought of as a sequence of lines, much like a Python

string can be thought of as a sequence of characters.

* To break the file into lines, there is a special character that represents the

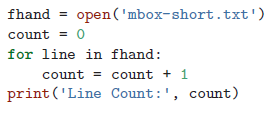
“end of the line” called the newline character.

* we represent the newline character as a backslash-n in string constants.
* Even though this looks like two characters, it is actually a single character.
* 

## Reading files

* While the file handle does not contain the data for the file, it is quite easy to

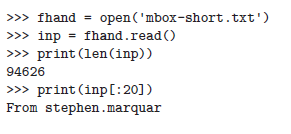
construct a for loop to read through and count each of the lines in a file:

* 
* For loop causes data to be read from the file
* When the file is read using a for loop in this manner, Python takes care of

splitting the data in the file into separate lines using the newline character.

* The above program can count the lines in any size file using very little

memory since each line is read, counted, and then discarded

* Another way:
* 

## Searching through a file

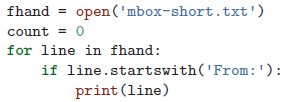
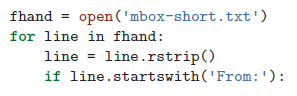
* When you are searching through data in a file, it is a very common pattern to

read through a file, ignoring most of the lines and only processing lines

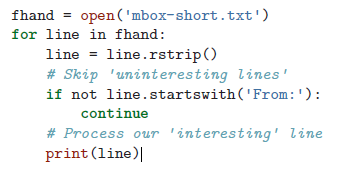
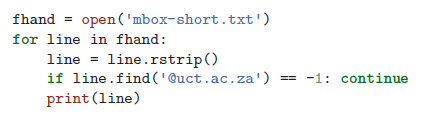
which meet a particular condition.

* We can combine the pattern for reading a file with string methods to build

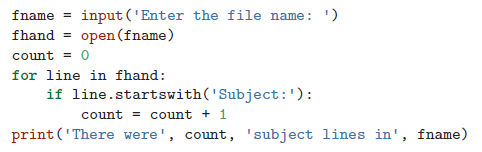
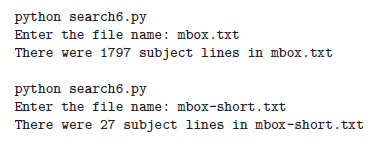
simple search mechanisms.

* 
* The Output will have extra blank line
* This is due to that invisible newline character. Print adds another new line character
* We could use line slicing to print all but the last character, but a simpler approach is to use the rstrip method which strips whitespace from the right side of a string as follows:
* 
* The basic idea of the search loop is that you are looking for “interesting”

lines and effectively skipping “uninteresting” lines.

* And then when we find an interesting line, we do something with that line.
* 
* Print line with a specific string in it
* 

## Letting the user choose the file name

* 
* 

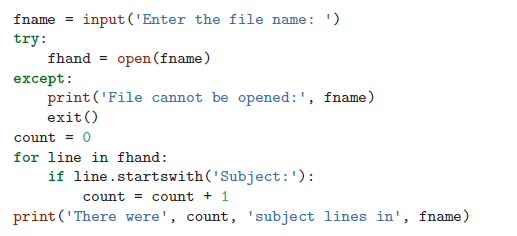
## Try, Except and open

* What if our user types something that is not a file name?
* Users will eventually do every possible thing they can do to break your

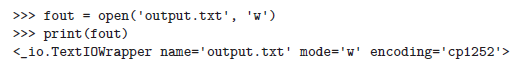
programs, either on purpose or with malicious intent.

* So now that we see the flaw in the program, we can elegantly fix it using the

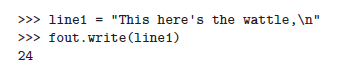
try/except structure.

* 
* The exit function terminates the program.
* It is a function that we call that never returns.

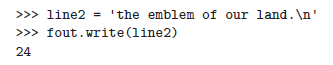
## Writing to Files

* To write a file, you have to open it with mode “w” as a second parameter:
* 
* If the file already exists, opening it in write mode clears out the old data and starts fresh.
* The write method of the file handle object puts data into the file, returning

the number of characters written.

* 
* The default write mode is text for writing (and reading) strings.
* Again, the file object keeps track of where it is, so if you call write again, it

adds the new data to the end.

* We must use explicit end of line character
* 
* After operation, close the handle using close() function
* 

# MODULE-3

# Lists

* A list is an ordered sequence of values. It is a data structure in Python. The values inside

the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are

called as elements or items. The elements of lists are enclosed within square brackets.

* ls1=[10,-4, 25, 13]
* ls2=[“Tiger”, “Lion”, “Cheetah”]
* We can have mixed type of elements in list. For example,
* ls3=[3.5, ‘Tiger’, 10, [3,4]]
* An empty list can be created any of the following ways –
* >>> ls =[]
* >>>ls = list()

## Lists are mutable

* The elements in the list can be accessed using a numeric index within square-brackets. It is

similar to extracting characters in a string.

>>> ls=[34, 'hi', [2,3],-5]

>>> print(ls[1])

hi

>>> print(ls[2])

[2, 3]

* To access the elements within inner list, we need to use double-indexing
* print(ls[2][0])
* The list can be thought of as a relationship between indices and elements. This relationship

is called as a mapping.

## Traversing a List

* A list can be traversed using for loop.
* for item in ls:

print(item)

## List Operations

* Python allows to use operators + and \* on lists. The operator + uses two list objects and

returns concatenation of those two lists. Whereas \* operator take one list object and one

integer value, say n, and returns a list by repeating itself for n times.

* >>> print(ls1+ls2) #concatenation using +

[1, 2, 3, 5, 6, 7

* print(ls1\*3) #repetition using \* [1, 2, 3, 1, 2, 3, 1, 2, 3]

## List Slices

* Similar to strings, the slicing can be applied on lists as well.
* >>> print(t[:])
* ['a', 'b', 'c', 'd', 'e']
* >>> print(t[1:])
* ['b', 'c', 'd', 'e']
* >>> print(t[:3])
* ['a', 'b', 'c']
* >>> print(t[2:4])
* ['c', 'd']
* >>> print(t[:-2])
* ['a', 'b', 'c']
* print(t[::-1])
* ['e', 'd', 'c', 'b', 'a']
* >>> t[1:3]=['p','q']
* >>> print(t)
* ['a', 'p', 'q', 'd', 'e']

## List Methods

**append():** This method is used to add a new element at the end of a list.

>>> ls=[1,2,3]

>>> ls.append(‘hi’)

>>> ls.append(10)

>>> print(ls)

[1, 2, 3, ‘hi’, 10]

**extend():** This method takes a list as an argument and all the elements in this list

are added at the end of invoking list.

>>> ls1=[1,2,3]

>>> ls2=[5,6]

>>> ls2.extend(ls1)

>>> print(ls2)

[5, 6, 1, 2, 3]

Now, in the above example, the list ls1 is unaltered.

sort(): This method is used to sort the contents of the list. By default, the function

will sort the items in ascending order.

>>> ls=[3,10,5, 16,-2]

>>> ls.sort()

>>> print(ls)

[-2, 3, 5, 10, 16]

When we want a list to be sorted in descending order, we need to set the argument

as shown –

>>> ls.sort(reverse=True)

>>> print(ls) [16, 10, 5, 3, -2]

**count():** This method is used to count number of occurrences of a particular value

within list.

>>> ls=[1,2,5,2,1,3,2,10]

>>> ls.count(2)

3 #the item 2 has appeared 3 tiles in ls

**clear():** This method removes all the elements in the list and makes the list empty.

>>> ls=[1,2,3]

>>> ls.clear()

>>> print(ls)

**insert():** Used to insert a value before a specified index of the list.

>>> ls=[3,5,10]

>>> ls.insert(1,"hi")

>>> print(ls)

[3, 'hi', 5, 10]

**index():** This method is used to get the index position of a particular value in the list.

>>> ls=[4, 2, 10, 5, 3, 2, 6]

>>> ls.index(2)

1

## Deleting elements from the list

**pop():** This method deletes the last element in the list, by default.

>>> ls=[3,6,-2,8,10]

>>> x=ls.pop() #10 is removed from list and stored in x

>>> print(ls)

[3, 6, -2, 8]

>>> print(x)

10

**remove():** When we don’t know the index, but know the value to be removed, then

this function can be used.

>>> ls=[5,8, -12,34,2]

>>> ls.remove(34)

>>> print(ls)

[5, 8, -12, 2]

Unlike pop() function, the remove() function will not return the value that has been

deleted.

**del:** This is an operator to be used when more than one item to be deleted at a time.

Here also, we will not get the items deleted.

>>> ls=[3,6,-2,8,1]

>>> del ls[2] #item at index 2 is deleted

>>> print(ls)

[3, 6, 8, 1]

## Lists and functions and strings

* The utility functions like max(), min(), sum(), len() etc. can be used on lists.
* Though both lists and strings are sequences, they are not same. In fact, a list of characters

is not same as string.

* To convert a string into a list, we use a method list() as below –
* >>> s="hello"
* >>> ls=list(s)
* >>> print(ls)
* ['h', 'e', 'l', 'l', 'o']
* no argument is provided, the split() function takes the delimiter as whitespace.
* >>> dt="20/03/2018"
* >>> ls=dt.split('/')
* >>> print(ls)
* ['20', '03', '2018']
* Join() has opposite behaviour from split
* >>> ls=["Hello", "how", "are", "you"]
* >>> d=' '
* >>> d.join(ls)
* 'Hello how are you

## Parsing Lines

* In many situations, we would like to read a file and extract only the lines containing required

pattern. This is known as parsing

From stephen.marquard@uct.ac.za Fri Jan 5 09:14:16 2018

From georgek@uct.ac.za Sat Jan 6 06:12:51 2018

fhand = open(‘logFile.txt’)

for line in fhand:

line = line.rstrip()

if not line.startswith('From '):

continue

words = line.split()

print(words[2])

## Objects and Values

* a= “hi” ; b= “hi”
* For strings, a and b point to same object
* >>> a= “hi”
* >>> b= “hi”
* >>> a is b #result is True
* >>> a==b #result is True

When two variables are referring to same object, they are called as identical objects.

When two variables are referring to different objects, but contain a same value, they are

known as equivalent objects. For example,

>>> s1=input(“Enter a string:”) #assume you entered hello

>>> s2= input(“Enter a string:”) #assume you entered hello

>>> s1 is s2 #check s1 and s2 are identical

False

>>> s1 == s2 #check s1 and s2 are equivalent

True

* String literals are interned by default. That is, when two string literals are created in the

program with a same value, they are going to refer same object.

* >>> ls1=[1,2,3]
* >>> ls2=[1,2,3]
* >>> ls1 is ls2 #output is False
* >>> ls1 == ls2 #output is True

## Aliasing

When an object is assigned to other using assignment operator, both of them will refer to

same object in the memory. The association of a variable with an object is called as

reference.

>>> ls1=[1,2,3]

>>> ls2= ls1

>>> ls1 is ls2 #output is True

Now, ls2 is said to be reference of ls1. In other words, there are two references to the

same object in the memory.

An object with more than one reference has more than one name, hence we say that object

is aliased. If the aliased object is mutable, changes made in one alias will reflect the other.

>>> ls2[1]= 34

>>> print(ls1) #output is [1, 34, 3]

## List as function arguments

* def del\_front(t):
* del t[0]
* ls = ['a', 'b', 'c']
* del\_front(ls)
* print(ls) # output is ['b', 'c']
* For example, the append() function modifies the list, whereas the +
* operator creates a new list.
* >>> t1 = [1, 2]
* >>> t2 = t1.append(3)
* >>> print(t1) #output is [1 2 3]
* >>> print(t2) #prints None
* >>> t3 = t1 + [5]
* >>> print(t3) #output is [1 2 3 5]
* >>> t2 is t3 #output is False
* def test(t):
* t=t[1:]
* ls=[1,2,3]
* test(ls)
* print(ls) #prints [1, 2, 3]

# Dictionaries

* A dictionary is a key value pair.
* It a mapping between set of indices and set of values
* Each key maps a value
* Eng2sp=dict()
* Eng2sp[‘One’] =’Uno’
* Eng2sp = {‘one’:’uno’, ‘two’:’dos}
* print (Eng2sp[‘One])
* Eng2sp.keys() returns list of keys
* Eng2sp.values() returns list of values
* ‘one’ in eng2sp #True
* ‘uno’ in eng2sp #False

## Dictionary as set of counters

Assume that we need to count the frequency of alphabets in a given string. There are

different methods to do it –

* Create 26 variables to represent each alphabet. Traverse the given string and

increment the corresponding counter when an alphabet is found.

* Create a list with 26 elements (all are zero in the beginning) representing alphabets.
* Traverse the given string and increment corresponding indexed position in the list

when an alphabet is found.

* Create a dictionary with characters as keys and counters as values. When we find a

character for the first time, we add the item to dictionary. Next time onwards, we

increment the value of existing item.

string=input("Enter a string ")

d=dict()

for i in string:

if(i not in d):

d[i]=1

else :

d[i]+=1

print(d)

## Get Method

* It takes a key and a default value
* If key is in the dictionary, it returns a corresponding value
* Otherwise it returns default value
* E.g.: print(counts.get(‘jan’,0)) #counts is the dictionary
* If jan is present in the dictionary, it returns corresponding value else it returns default value

s=input("Enter a string:")

d=dict()

for ch in s:

d[ch]=d.get(ch,0)+1

print(d)

## Looping and Dictionaries

* When a for-loop is applied on dictionaries, it will iterate over the keys of dictionary. If we

want to print key and values separately, we need to use the statements as shown –

tel\_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}

for k in tel\_dir:

print(k, tel\_dir[k])

* Note that, while accessing items from dictionary, the keys may not be in order. If we want to

print the keys in alphabetical order, then we need to make a list of the keys, and then sort

that list. We can do so using keys() method of dictionary and sort() method of lists

tel\_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}

ls=list(tel\_dir.keys())

print("The list of keys:",ls)

ls.sort()

print("Dictionary elements in alphabetical order:")

for k in ls:

print(k, tel\_dir[k])

* The key-value pair from dictionary can be together accessed with the help of a

method items() as shown –

>>> d={'Tom':3412, 'Jerry':6781, 'Mickey':1294}

>>> for k,v in d.items():

print(k,v)

## Dictionaries and Files

* A dictionary can be used to count the frequency of words in a file.
* we need to take an outer loop for iterating over entire file, and an inner loop for traversing each line in a file. Then in every line, we count the occurrence of a word, as we did before for a character. The program is given as below

f=open("a.txt")

d=dict()

for i in f:

for word in i.split():

d[word]=d.get(word,0) +1

print(d)

## Looping and Dictionaries

* If you use for loop to traverse a dictionary, it traverses the keys of dictionary

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}

for key in counts:

print(key, counts[key])

counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}

for key in counts:

if counts[key] > 10 :

print(key, counts[key])

## Advanced Text Parsing

* The string module of Python provides a list of all punctuation marks as shown –

>>> import string

>>> string.punctuation

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

* The str class has a method maketrans() which returns a translation table usable for another

method translate(). Consider the following syntax to understand it more clearly –

line.translate(str.maketrans(fromstr, tostr, deletestr))

The above statement replaces the characters in fromstr with the character in the same

position in tostr and delete all characters that are in deletestr. The fromstr and

tostr can be empty strings and the deletestr parameter can be omitted.

* We can rewrite the program to remove punctuation as shown below.

import string

fname=input("Enter file name:")

try:

fhand=open(fname)

except:

print("File cannot be opened")

exit()

d=dict()

for line in fhand:

line=line.rstrip()

line=line.translate(line.maketrans('','',string.punctuation))

line=line.lower()

for word in line.split():

d[word]=d.get(word,0)+1

print(d)

## Debugging

* When we are working with big datasets (like file containing thousands of pages), it is

difficult to debug by printing and checking the data by hand. So, we can follow any of the

following procedures for easy debugging of the large datasets –

**Scale down the input:** If possible, reduce the size of the dataset.

**Check summaries and types:** Instead of printing and checking the entire dataset,

consider printing summaries of the data.

**Write self-checks:** Sometimes you can write code to check for errors automatically. For

example, if you are computing the average of a list of numbers, you could check that the

result is not greater than the largest element in the list or less than the smallest. This is

called a sanity check because it detects results that are “completely illogical”. Another

kind of check compares the results of two different computations to see if they are

consistent. This is called a consistency check.

**Pretty print the output:** Formatting debugging output can make it easier to spot an

error.

# Tuples

* A tuple is a sequence of items, similar to lists. The values stored in the tuple can be of any

type and they are indexed using integers. Unlike lists, tuples are immutable.

* Tuples are comparable and hashable objects. Hence, they can be made as keys in dictionaries.
* A tuple can be created in Python as a comma separated list of items – may or may not be

enclosed within parentheses.

>>> t='Mango', 'Banana', 'Apple' #without parentheses

>>> print(t)

('Mango', 'Banana', 'Apple')

>>> t1=('Tom', 341, 'Jerry') #with parentheses

>>> print(t1)

('Tom', 341, 'Jerry')

* If we would like to create a tuple with single value, then just a parenthesis will not suffice. We need to include a comma as well.

>>> t=3, #or use the statement t=(3,)

>>> type(t) #now this is a tuple

<class 'tuple'>

* An empty tuple can be created either using a pair of parenthesis or using a function tuple()
* If we provide an argument of type sequence (a list, a string or tuple) to the method tuple(), then a tuple with the elements in a given sequence will be created
* >>> t=tuple('Hello')
* >>> print(t)
* ('H', 'e', 'l', 'l', 'o')

## Comparing Tuples

* Tuples can be compared using operators like >, <, >=, == etc.
* The meaning of < and > in tuples is not exactly less than and greater than, instead, it

means comes before and comes after. Hence in such cases, we will get results different

from checking equality (==).

>>> (1,2,3)<(1,2,5)

True

>>> (3,4)<(5,2)

True

* When we use relational operator on tuples containing non-comparable types, then

TypeError will be thrown.

* The sort() function internally works on similar pattern – it sorts primarily by first element, in

case of tie, it sorts on second element and so on. This pattern is known as DSU (Decorate Sort and Undecorate)

* Consider a program of sorting words in a sentence from longest to shortest, which

illustrates DSU property.

txt = 'Ram and Seeta went to forest with Lakshman'

words = txt.split()

t = list()

for word in words:

t.append((len(word), word))

print(‘The list is:’,t)

t.sort(reverse=True)

res = list()

for length, word in t:

res.append(word)

print(‘The sorted list:’,res)

* The output would be –

The list is: [(3, 'Ram'), (3, 'and'), (5, 'Seeta'), (4, 'went'),

(2, 'to'), (6, 'forest'), (4, 'with'), (8, 'Lakshman')]

The sorted list: ['Lakshman', 'forest', 'Seeta', 'went', 'with',

'and', 'Ram', 'to']

## Tuple Assignments

* Tuple has a unique feature of having it at LHS of assignment operator. This allows us to

assign values to multiple variables at a time.

>>> x,y=10,20

>>> print(x) #prints 10

>>> print(y) #prints 20

* The best known example of assignment of tuples is swapping two values as below –

>>> a=10

>>> b=20

>>> a, b = b, a

>>> print(a, b) #prints 20 10

* Giving more values than variables generates ValueError –

>>> a, b=10,20,5

ValueError: too many values to unpack (expected 2)

## Dictionaries and Tuples

* Dictionaries have a method called items() that returns a list of tuples, where each tuple is a

key-value pair as shown below –

>>> d = {'a':10, 'b':1, 'c':22}

>>> t = list(d.items())

>>> print(t)

[('b', 1), ('a', 10), ('c', 22)]

* As dictionary may not display the contents in an order, we can use sort() on lists
* We can combine the method items(), tuple assignment and a for-loop to get a pattern for

traversing dictionary:

d={'Tom': 1292, 'Jerry': 3501, 'Donald': 8913}

for key, val in list(d.items()):

print(val,key)

The output would be –

1292 Tom

3501 Jerry

8913 Donald

## The Most Common Words

* Write a program to find most commonly used words in a text file
* Remove all punctuations , construct dictionary, make a list and sort, print first 10

import string

fhand = open('test.txt')

counts = dict()

for line in fhand:

line = line.translate(str.maketrans('', '',string.punctuation))

line = line.lower()

for word in line.split():

if word not in counts:

counts[word] = 1

else:

counts[word] += 1

lst = list()

for key, val in list(counts.items()):

lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:

print(key, val)

## Using Tuples as Keys in Dictionaries

* when we want a dictionary containing composite keys, we will use tuples. For Example, we may need to create a telephone directory where name of a person is Firstname-last name pair and value is the telephone number.

Consider the program to do this task –

names=(('Tom','Cat'),('Jerry','Mouse'), ('Donald', 'Duck'))

number=[3561, 4014, 9813]

telDir={}

for i in range(len(number)):

telDir[names[i]]=number[i]

for fn, ln in telDir:

print(fn, ln, telDir[fn,ln])

The output would be –

Tom Cat 3561

Jerry Mouse 4014

Donald Duck 9813

## Debugging

* Lists, Dictionaries and Tuples are basically data structures. In real-time programming, we

may require compound data structures like lists of tuples, dictionaries containing tuples and

lists etc. But, these compound data structures are prone to shape errors

## Regular Expressions

* Searching for required patterns and extracting only the lines/words matching the pattern is

a very common task in solving problems programmatically. We have done such tasks

earlier using string slicing and string methods like split(), find() etc. As the task of searching

and extracting is very common, Python provides a powerful library called regular

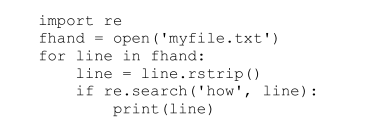
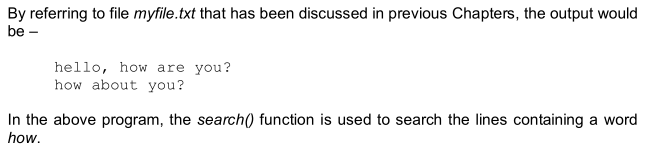
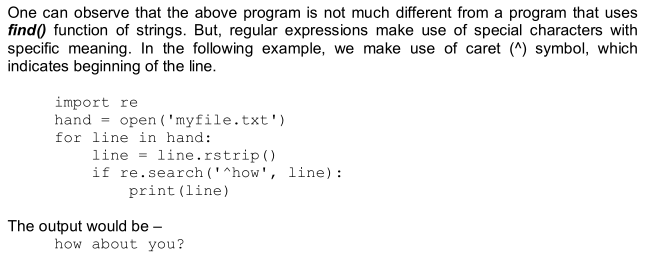
expressions to handle these tasks elegantly. Though they have quite complicated syntax,

they provide efficient way of searching the patterns.

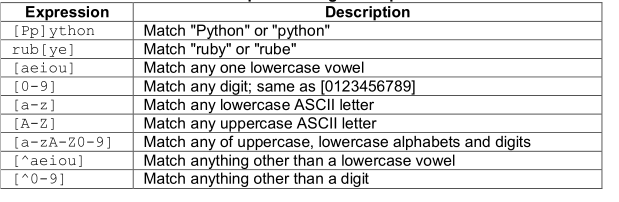
* The regular expressions are themselves little programs to search and parse strings. To use

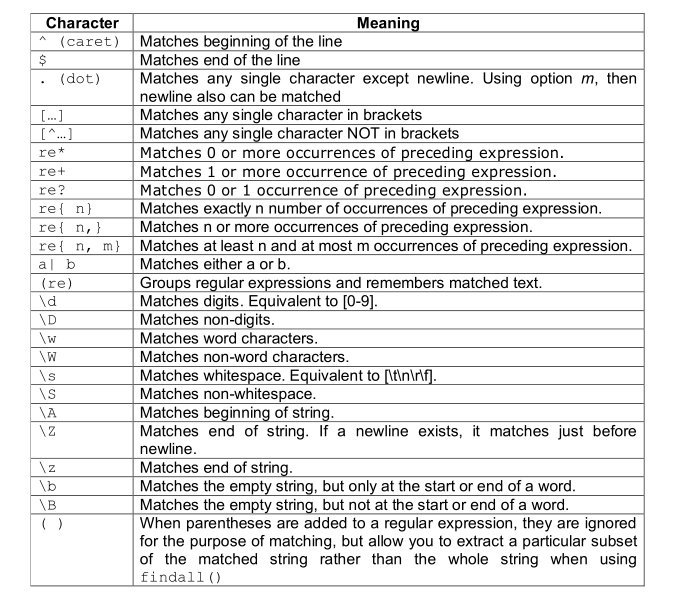
them in our program, the library/module re must be imported. There is a search() function

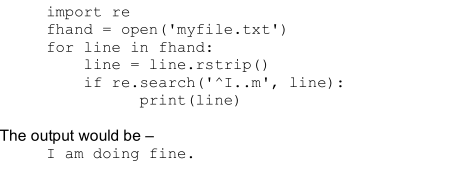
in this module, which is used to find particular substring within a string.

* 
* 
* 

## Character Matching in Regular Expressions

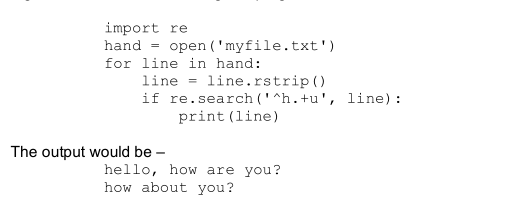
* Python provides a list of meta-characters to match search strings.
* 

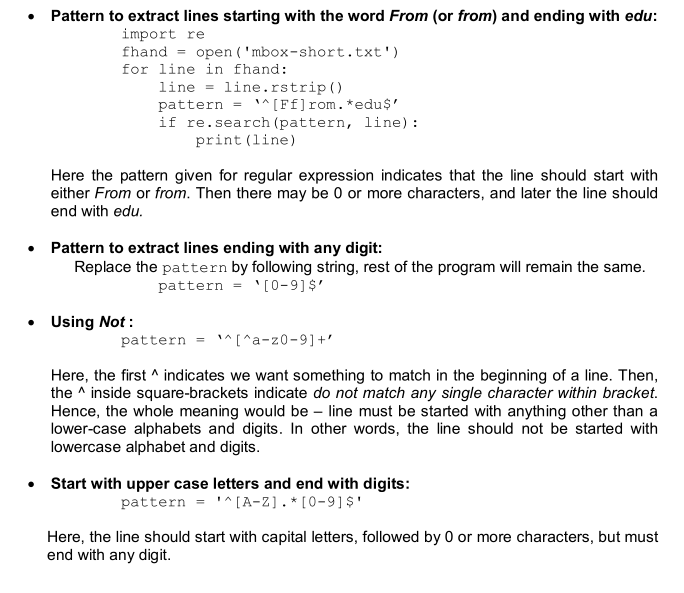


* Most commonly used metacharacter is dot, which matches any character.
* 
* In the previous program, we knew that there are exactly two characters between I and m.

Hence, we could able to give two dots. But, when we don’t know the exact number of

characters between two characters (or strings), we can make use of dot and + symbols.

* 

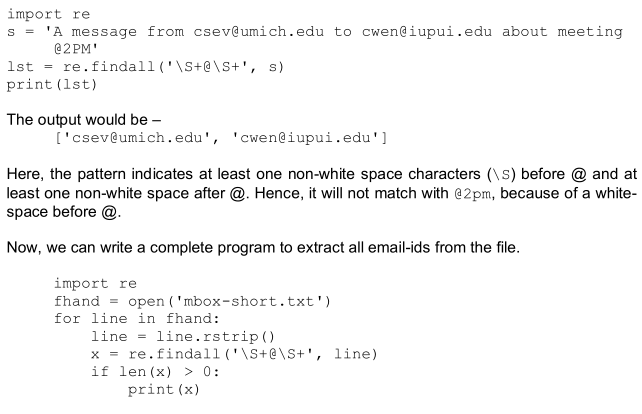


## Extracting Data using Regular Expressions

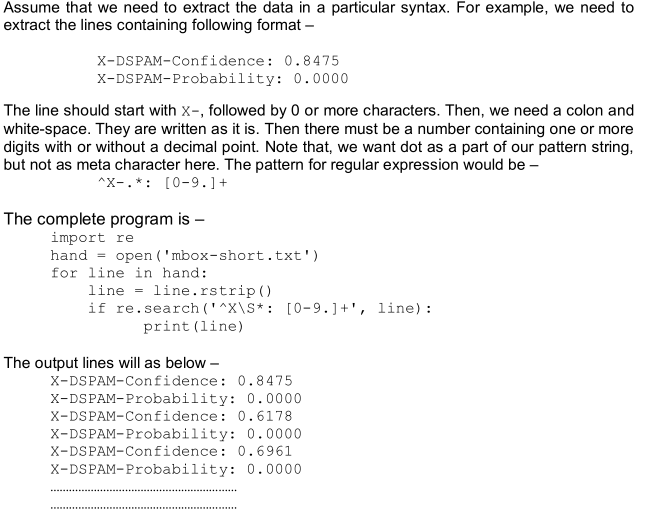
* Python provides a method findall() to extract all of the substrings matching a regular

expression. This function returns a list of all non-overlapping matches in the string. If there

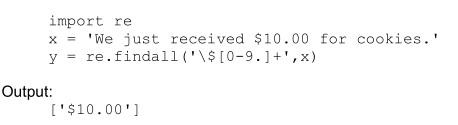
is no match found, the function returns an empty list.



## Combining Searching and Extracting

* 

## Escape Character

* Sometimes, we need meta characters themselves as a part of matching string. Then, we need to escape them using a back slash
* 

## Bonus Section for Unix/Linux Users

* There is a command-line program built into Unix called grep (Generalized Regular Expression Parser) that behaves similar to search() function.

$ grep '^From:' mbox-short.txt

Output:

From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

# MODULE 4

## Classes and Objects

* Class is a user-defined data type which binds data and functions together into single entity. Class is just a prototype (or a logical entity/blue print) which will not consume any memory.
* An object is an instance of a class and it has physical existence. One can create any number of objects for a class.
* A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).

## Programmer-defined Types

* A class in Python can be created using a keyword class.
* The keyword pass can be used when you don’t need any class members within it.

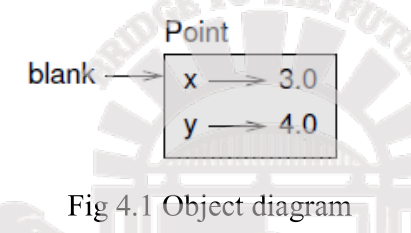
class Point:

pass

print(Point)

* Defining a class named Point creates a class object
* >>> Point
* The output would be – <class '\_\_main\_\_.Point'>
* The term \_\_main\_\_ indicates that the class Point is at the top level while executing the program.
* you can create any number of objects of this class.
* >>> blank = Point()
* >>> blank
* <\_\_main\_\_.Point object at 0x003C1BF0>
* Creating a new object is called instantiation and the object is an instance of the class.
* When you print an instance, Python tells you what class it belongs to and where it is stored in memory

## Attributes

* An object can contain named elements known as attributes. One can assign values to these attributes(instance) using dot operator.
* The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number
* The syntax for two attributes x and y for the object blank of a class Point is as below
* blank.x = 10.0
* blank.y = 20.0
* A state diagram that shows an object and its attributes is called as object diagram
* 
* One can access attributes of an object as shown –
* >>> print(blank.x) #10.0
* >>> print(blank.y) #20.0

Program: Write a class Point representing a point on coordinate system. Implement following functions

* A function read\_point() to receive x and y attributes of a Point object as user input
* A function distance() which takes two objects of Point class as arguments and computes the Euclidean distance between them.
* A function print\_point() to display one point in the form of ordered-pair.

import math

class Point:

""" This is a class Point representing a

coordinate Point

"""

def read\_point(p):

p.x=float(input("x coordinate"))

p.y=float(input("y coordinate:"))

def print\_point(p):

print("(%g,%g)"%(p.x, p.y))

def distance(p1,p2):

d=math.sqrt((p1.x-p2.x)\*\*2+(p1.y-p2.y)\*\*2)

return d

p1=Point() #create first object

print("Enter First point:")

#read x and y for p1 read\_point(p1)

p2=Point() #create second object

print("Enter Second point:")

#read x and y for p2 read\_point(p2)

dist=distance(p1,p2) #compute distance

print("First point is:")

#print p1 print\_point(p1)

print("Second point is:")

#print p2 print\_point(p2)

print("Distance is: %g" %(distance(p1,p2)))

## Rectangles Program

* It is possible to make an object of one class as an attribute to other class

Program:Write a class Rectangle containing numeric attributes width and height. This class should contain another attribute corner which is an instance of another class Point. Implement following functions –

A function to print corner point as an ordered-pair

A function find\_center() to compute center point of the rectangle

A function resize() to modify the size of rectangle

class Point:

""" This is a class Point representing

coordinate point """

class Rectangle:

""" This is a class Rectangle.

Attributes: width, height and Corner Point """

def find\_center(rect):

p=Point()

p.x = rect.corner.x + rect.width/2

p.y = rect.corner.y + rect.height/2

return p

def resize(rect,w, h):

rect.width +=w

rect.height +=h

def print\_point(p):

print("(%g,%g)"%(p.x, p.y))

box=Rectangle()

box.corner=Point()

box.width=100

box.height=200

box.corner.x=0

box.corner.y=0

print("Original Rectangle is:")

print("width=%g, height=%g"%(box.width, box.height))

center=find\_center(box)

print("The center of rectangle is:")

print\_point(center)

resize(box,50,70)

print("Rectangle after resize:")

print("width=%g, height=%g"%(box.width, box.height))

center=find\_center(box)

print("The center of resized rectangle is:")

print\_point(center)

## Objects are mutable

* The function resize() in the above program, takes three arguments: rect – an instance of Rectangle class and two numeric variables w and h.
* The values w and h are added to existing attributes width and height. This clearly shows that objects are mutable.
* State of an object can be changed by modifying any of its attributes.
* When this function is called with a statement – resize(box,50,70) the rect acts as an alias for box.
* Thus, the above program illustrates the concepts: object of one class is made as attribute for object of another class, returning objects from functions and objects are mutable.

## Copying

* An object will be aliased whenever an object is assigned to another object of same class.

>>> class Point:

pass

>>> p1=Point()

>>> p1.x=3

>>> p1.y=4

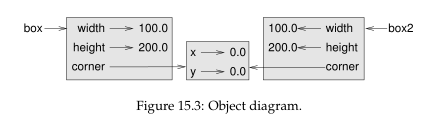
>>> p2=p1

>>> print(p1)

<\_\_main\_\_.Point object at 0x01581BF0>

>>> print(p2)

<\_\_main\_\_.Point object at 0x01581BF0>

* 
* >>> p1 is p2 True
* >>> p1==p2 True

>>> p1 = Point()

>>> p1.x = 3.0

>>> p1.y = 4.0

>>> import copy

>>> p2 = copy.copy(p1)

p1 and p2 contain the same data, but they are not the same Point.

>>> print\_point(p1)

(3, 4)

>>> print\_point(p2)

(3, 4)

>>> print(p1)

<\_\_main\_\_.Point object at 0x01581BF0>

>>> print(p3)

<\_\_main\_\_.Point object at 0x02344A50>

>>> print(p3.x,p3.y)

3.0 4.0

>>> p1 is p2

False

>>> p1 == p2

False

Shallow Copy

>>> box2 = copy.copy(box1)

>>> box2 is box1

False

>>> box2.corner is box1.corner

True

Deep Copy

>>> box2 = copy.deepcopy(box1)

>>> box2 is box1

False

>>> box2.corner is box1.corner

False

* we want two independent physical objects. Python provides a method deepcopy() for doing this task.
* In the above example, the physical object to which box1.corner and box2.corner are pointing is only one. This type of copying the objects is known as shallow copy.

## Classes and Functions

* Python presents functional programming style and two types of functions i.e pure functions and modifiers.

## Pure Functions

* Pure functions is one that does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.
* let us consider an example of creating a class called Time. An object of class Time contains hour, minutes and seconds as attributes.
* Define a class called Time that records the time of day. write a function called print\_time that takes a Time object and prints it in the form hour:minute:second. Add two time objects that yield proper result and hence need to check whether number of seconds exceeds 60, minutes exceeds 60 etc, and take appropriate action.
* Here, the function add\_ time() takes two arguments of type Time, and returns a Time object, whereas, it is not modifying contents of its arguments t1 and t2. Such functions are called as pure functions.

class Time:

"""Represents the time of a day Attributes:

hour, minute, second """

def printTime(t):

print("%.2d:%.2d:%.2d"%(t.hour,t.minute,t.second))

def add\_time(t1,t2):

sum=Time()

sum.hour = t1.hour + t2.hour

sum.minute =t1.minute + t2.minute

sum.second =t1.second + t2.second

if sum.second >= 60:

sum.second -= 60

sum.minute += 1

if sum.minute >= 60:

sum.minute -= 60

sum.hour += 1

return sum

t1=Time()

t1.hour=10

t1.minute=34

t1.second=25

print("Time1 is:")

printTime(t1)

t2=Time()

t2.hour=2

t2.minute=12

t2.second=41

print("Time2 is :")

printTime(t2)

t3=add\_time(t1,t2)

print("After adding two time objects:")

printTime(t3)

Output:

Time1 is: 10:34:25

Time2 is : 02:12:41

After adding two time objects: 12:47:06

## Modifiers

* Sometimes it is useful for a function to modify the objects passed as parameters. In that case, the changes are visible to the caller and the parameter value changes during function execution. Functions that work this way are called modifiers
* Eg: A function Increment(), which adds a given number of seconds to a Time object

def increment(t,seconds):

t.second += seconds

while t.second >= 60:

t.second -= 60

t.minute += 1

while t.minute >= 60:

t.minute -= 60

t.hour += 1

* In this function, we initially add the argument seconds to t.second. Now, there is a chance that t.second is exceeding 60. So, we will increment minute counter till t.second becomes lesser than 60. Similarly, till the t.minute becomes lesser than 60, we will decrement minute counter.

## Prototyping v/s Planning

* When a problem statement is not known completely , we may write the program initially, and then keep modifying it as and when requirement (problem definition) changes. This methodology is known as prototype and patch.
* An alternative is designed development, in which high-level insight into the problem can make the programming much easier.
* In add\_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem. we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

def time\_to\_int(time):

minutes = time.hour \* 60 + time.minute

seconds = minutes \* 60 + time.second

return seconds

def int\_to\_time(seconds):

time = Time()

minutes, time.second = divmod(seconds, 60)

time.hour, time.minute = divmod(minutes, 60)

return time

## CLASSES AND METHODS

* The classes considered so far were empty classes without having any definition. But, in a true object-oriented programming, a class contains class-level attributes, instance-level attributes, methods etc.

## OBJECT ORIENTED FEATURES

* Programs include class and method definitions.
* Most of the computation is expressed in terms of operations on objects.
* Objects often represent things in the real world, and methods often correspond to the ways objects in the real world interact
* To establish strong relationship between the object of the class and a function, we must define a function as a member of the class. A function which is associated with a particular class is known as a method.
* Methods are semantically the same as functions, but there are two syntactic differences:

1. Methods are defined inside a class definition in order to make the relationship between the class and the method explicit
2. The syntax for invoking a method is different from the syntax for calling a function.

## THE \_\_init\_\_() METHOD

* The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated.
* Its full name is \_\_init\_\_
* An init method for the Time class might look like this:

class Time:

“”” “””

def \_\_init\_\_(self, hour=0, minute=0, second=0):

self.hour = hour

self.minute = minute

self.second = second

* It is common for the parameters of \_\_init\_\_ to have the same names as the attributes. The statement self.hour = hour stores the value of the parameter hour as an attribute of self.

The parameters are optional, so if you call Time with no arguments, you get the default values.

>>> time = Time()

>>> time.print\_time()

00:00:00

>>> time = Time (9)

>>> time.print\_time()

09:00:00

If you provide two arguments, they override hour and minute.

>>> time = Time(9, 45)

>>> time.print\_time()

09:45:00

And if you provide three arguments, they override all three default values.

## THE \_\_str() METHOD

* \_\_str\_\_ is a special method, like \_\_init\_\_, that is supposed to return a string representation of an object

For example, here is a str method for Time objects:

class Time:

“”” “””

def \_\_str\_\_(self):

return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)

When you print an object, Python invokes the str method:

>>> time = Time(9, 45)

>>> print(time)

09:45:00

* It is a good practice to always start by writing \_\_init\_\_, which makes program easier to instantiate objects, and \_\_str\_\_, which is useful for debugging.

this code is example for \_\_str\_\_

class Student:

def \_\_init\_\_(self,rollNo,firstName,lastName):

self.rollNo = rollNo

self.firstname = firstName

self.lastname = lastName

def \_\_str\_\_(self):

#convert the int to string

s\_rollNo = str(self.rollNo)

return s\_rollNo+":"+self.firstname+" "+self.lastname

s= Student(34,"chaitra"," rao")

print(s)

import math

class Point:

def \_\_init\_\_(self,a,b):

self.x=a

self.y=b

def dist(self,p2):

d=math.sqrt((self.x-p2.x)\*\*2 + (self.y-p2.y)\*\*2) return d

def \_\_str\_\_(self):

return "(%d,%d)"%(self.x, self.y)

p1=Point(10,20) #\_\_init\_\_() is called automatically

p2=Point(4,5) #\_\_init\_\_() is called automatically

print("P1 is:",p1) #\_\_str\_\_() is called automatically

print("P2 is:",p2) #\_\_str\_\_() is called automatically

d=p1.dist(p2) #explicit call for dist()

print("The distance is:",d)

The sample output is –

P1 is: (10,20)

P2 is: (4,5)

Distance is: 16.15549442140351

Program explaination:

Every method of any class must have the first argument as self. By convention, the

first parameter of a method is called self. The argument self is a reference to the

current object. That is, it is reference to the object which invoked the method.

class Time:

def print\_time(self):

print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))

The reason for this convention is an implicit metaphor:

• The syntax for a function call, print\_time(start), suggests that the function is the

active agent.

• In object-oriented programming, the objects are the active agents. A method

invocation will be start.print\_time()This change in perspective might be more polite,

but it is not obvious that it is useful. In the

examples we have seen so far, it may not be. But sometimes shifting responsibility

from the

functions onto the objects makes it possible to write more versatile functions (or

methods),

and makes it easier to maintain and reuse code.

the \_\_init\_\_() method will be called automatically. The internal meaning of

the above line is –

p1.\_\_init\_\_(10,20)

Here, p1 is the object which is invoking a method. Hence, reference to this object is

created and passed to \_\_init\_\_() as self. The values 10 and 20 are passed to formal

parameters a and b of \_\_init\_\_() method. Now, inside \_\_init\_\_() method, we have

statements

self.x=10

self.y=20

## OPERATOR OVERLOADING

* POLYMORPHISM (Poly= many; Morph = form) : Refers to the ability of an object/method to behave differently in different situations.
* Basic operators like +, -, \* etc. can be overloaded. To overload an operator, one needs to write a method within user-defined class. Python provides a special set of methods which is to be used for overloading required operator.
* When + operator is applied to Time objects, Python invokes \_\_add\_\_method When you print the result, Python invokes \_\_str\_\_, Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading.

class Time:

def \_\_init\_\_(self, h=0,m=0,s=0):

self.hour=h

self.min=m

self.sec=s

def time\_to\_int(self):

minute=self.hour\*60+self.min

seconds=minute\*60+self.sec return

seconds

def int\_to\_time(self, seconds):

t=Time()

minutes, t.sec=divmod(seconds,60)

t.hour, t.min=divmod(minutes,60) return t

def \_\_str\_\_(self):

return "%.2d:%.2d:%.2d"%(self.hour,self.min,self.sec)

def \_\_eq\_\_(self,t):

return self.hour==t.hour and self.min==t.min and self.sec==t.sec

def \_\_add\_\_(self,t):

if isinstance(t, Time): return

self.addTime(t)

else:

return self.increment(t)

def addTime(self, t): seconds=self.time\_to\_int()+t.time

\_to\_int() return self.int\_to\_time(seconds)

def increment(self, seconds): seconds +=

self.time\_to\_int() return

self.int\_to\_time(seconds)

def \_\_radd\_\_(self,t): return

self.\_\_add\_\_(t)

T1=Time(3,40)

T2=Time(5,45)

print("T1 is:",T1)

print("T2 is:",T2)

#call for \_\_eq\_\_() print("Whether T1 is same as T2?",T1==T2)

T3=T1+T2 #call for \_\_add\_\_()

print("T1+T2 is:",T3)

T4=T1+75 #call for \_\_add\_\_()

print("T1+75=",T4)

T5=130+T1 #call for \_\_radd\_\_()

print("130+T1=",T5)

T6=sum([T1,T2,T3,T4])

print("Using sum([T1,T2,T3,T4]):",T6)

The output would be –

T1 is: 03:40:00

T2 is: 05:45:00

Whether T1 is same as T2? False

T1+T2 is: 09:25:00

T1+75=

03:41:15

130+T1=

03:42:10

Using sum([T1,T2,T3,T4]): 22:31:15

# Module 5

## Networked Programs

### HTTP - Hypertext Transfer Protocol

* The dominant Application Layer Protocol on the Internet
* Invented for the Web - to Retrieve HTML, Images, Documents, etc.
* Communicates with TCP through sockets

## Getting Data From The Server

* Each time the user clicks on an anchor tag with an href= value to switch to a new page, the browser makes a connection to the web server and issues a “GET” request - to GET the content of the page at the specified URL
* The server returns the HTML document to the browser, which formats and displays the document to the user

## The world’s simplest web browser

**mysock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)**

**mysock.connect(('data.pr4e.org', 80))**

**cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()**

**mysock.send(cmd)**

**while True:**

**data = mysock.recv(512)**

**if (len(data) < 1):**

**break**

**print(data.decode(),end='')**

**mysock.close()**

**Output:**

HTTP/1.1 200 OK

Date: Sun, 14 Mar 2010 23:52:41 GMT

Server: Apache

Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT

ETag: "143c1b33-a7-4b395bea"

Accept-Ranges: bytes

Content-Length: 167

Connection: close

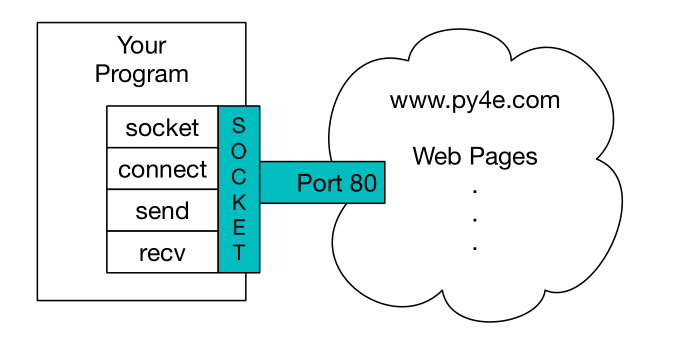
Content-Type: text/plain

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief



## Retrieving Images

import socket

import time

HOST = 'data.pr4e.org'

PORT = 80

mysock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

mysock.connect((HOST, PORT))

mysock.sendall(b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')

count = 0

picture = b""

while True:

data = mysock.recv(5120)

if len(data) < 1: break

#time.sleep(0.25)

count = count + len(data)

print(len(data), count)

picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)

pos = picture.find(b"\r\n\r\n")

print('Header length', pos)

print(picture[:pos].decode())

# Skip past the header and save the picture data

picture = picture[pos+4:]

fhand = open("stuff.jpg", "wb")

fhand.write(picture)

fhand.close()

## Retrieving web pages with urllib

Since HTTP is so common, we have a library that does all the socket work for us and makes web pages look like a file

import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

for line in fhand:

print(line.decode().strip())

But soft what light through yonder window breaks

It is the east and Juliet is the sun

Arise fair sun and kill the envious moon

Who is already sick and pale with grief

## Like a File Handler

import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()

for line in fhand:

words = line.decode().split()

for word in words:

counts[word] = counts.get(word, 0) + 1

print(counts)

## Reading Binary Files

import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg').read()

fhand = open('cover3.jpg', 'wb')

fhand.write(img)

fhand.close()

## Parsing HTML (a.k.a. Web Scraping)

* When a program or script pretends to be a browser and retrieves web pages, looks at those web pages, extracts information, and then looks at more web pages
* Search engines scrape web pages - we call this “spidering the web” or “web crawling”

#### Using Regular Expressions

<h1>The First Page</h1>

<p>

If you like, you can switch to the

<a href="http://www.dr-chuck.com/page2.htm">

Second Page</a>.

</p>

* href="http[s]?://.+?“

import urllib.request, urllib.parse, urllib.error

import re

import ssl

# Ignore SSL certificate errors

ctx = ssl.create\_default\_context()

ctx.check\_hostname = False

ctx.verify\_mode = ssl.CERT\_NONE

url = input('Enter - ')

html = urllib.request.urlopen(url).read()

links = re.findall(b'href="(http[s]?://.\*?)"', html)

for link in links:

print(link.decode())

#### Beautiful Soup

import urllib.request, urllib.parse, urllib.error

from bs4 import BeautifulSoup

url = input('Enter - ')

html = urllib.request.urlopen(url).read()

soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags

tags = soup('a')

for tag in tags:

print(tag.get('href', None))

## Web Services

* Accessing and transferring data across the web can be done and processed in two most common formats. The following are the two common data interchange format across the web
* XML - Document style data
* JSON - Dictionaries ,lists and other data types

#### XML – Extensible Mark up Language

* XML is a standard for custom mark up languages for text documents. It is entirely made of text. The text based tags are called mark up.
* It is different from HTML since it uses customised tags for storing elements whereas HTML uses predefined tags.Hence it is called extensible.
* XML is a portable, open source language that allows programmers to develop applications that can be read by other applications. It is mainly used in webpages, where the data has a specific structure. XML creates a tree-like structure that is easy to interpret and supports a hierarchy.
* XML documents have sections, called elements, defined by a beginning and an ending tag. A tag is a markup construct that begins with < and ends with >.
* The characters between the start-tag and end-tag, if there are any, are the element's content. Elements can contain markup,including other elements, which are called "child elements".
* The largest, top-level element is called the root, which contains all other elements.
* Attributes are name–value pair that exist within a start-tag or empty-element tag. An XML attribute can only have a single value and each attribute can appear at most once on each element.

#### XML Parser Architectures

* The entire XML file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.
* The XML tree structure makes navigation, modification, and removal relatively simple.
* Python has a built in library, ElementTree, that has functions to read and manipulate XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ElementTree has two classes for this purpose –
* **ElementTree** -represents the whole XML document as a tree

Element represents a single node in this tree. Interactions with the whole document (reading

and writing to/from files) are usually done on the ElementTree level. Interactions with a single

XML element and its sub-elements are done on the Element level.

* **Element.find(match)**

Finds the first subelement matching match. match may be a tag name or path. Returns an

element instance or None.

* **Element.text()**

accesses the element’s text content

* **Element.get()**

accesses the element’s attributes:

get(key, default=None)

Gets the element attribute named key.

Returns the attribute value, or default if the attribute was not found.

**Python program to parse an XML file and print element text and attributes**

import xml.etree.ElementTree as ET

data = '''

<person>

<name>Chuck</name>

<phone type="intl">

+1 734 303 4456

</phone>

<email hide="yes"/>

</person>'''

tree = ET.fromstring(data) #creates an Elementinstance

print('Name:', tree.find('name').text)

print('Attr:', tree.find('email').get('hide'))

**Looping through nodes**

**An XML tree can have any number of nodes and they can be traversed to process each**

of them separately.

import xml.etree.ElementTree as ET

input = '''

<stuff>

<users>

<user x="2">

<id>001</id>

<name>Chuck</name>

</user>

<user x="7">

<id>009</id>

<name>Brent</name>

</user>

</users>

</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')

print('User count:', len(lst))

for item in lst:

print('Name', item.find('name').text)

print('Id', item.find('id').text)

print('Attribute', item.get("x"))

**JAVASCRIPT OBJECT NOTATION(JSON)**

* JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for

humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. The JSON format was inspired by the object and array format used in the JavaScript language

* JSON codes data as key- value pairs.
* JSON has the advantage that it maps directly to some combination of dictionaries and lists.
* And since nearly all programming languages have something equivalent to Python’s dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

**Parsing JSON**

* JSON can be constructed by nesting dictionaries (objects) and lists as needed. In this example,
* a list of users are represented where each user is a set of key-value pairs (i.e., a dictionary). So it’s a list of dictionaries.
* Python provides built in json library to parse the JSON and read through the data.

import json

data = '''

[

]'''

{ "id" : "001",

"x" : "2",

"name" : "Chuck" } ,

{ "id" : "009",

"x" : "7",

"name" : "Chuck" }

info = json.loads(data)

print('User count:', len(info))

for item in info:

print('Name', item['name'])

print('Id', item['id'])

print('Attribute', item['x'])

* json.loads() returns a Python list which we traverse with a for loop, and each item within that list is a Python dictionary.

## APIs

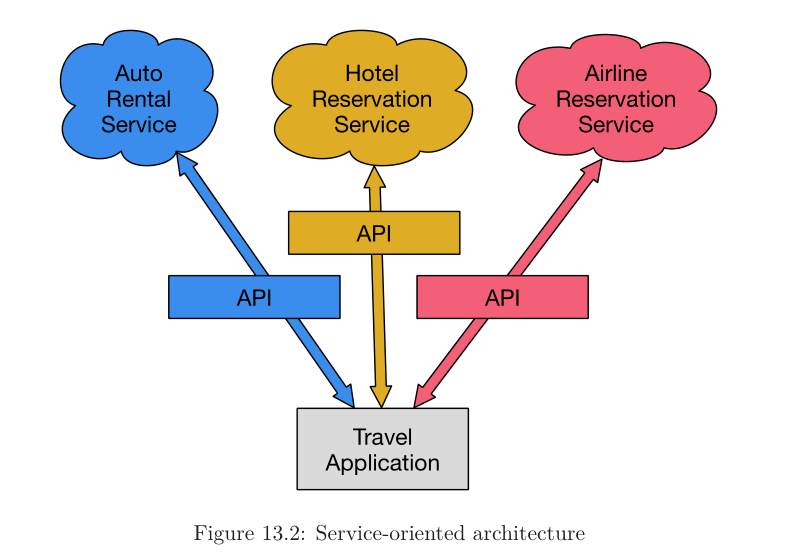
* Web service: When an application makes a set of services in its API available over the web, we call these web services.

#### API

* An application to application interface that’s allows data exchange between web services is called an Application Programming Interface.
* The general name for these application-to-application contracts is Application Program Interfaces or APIs.
* When we use an API, generally one program makes a set of services available for use by other applications and publishes the APIs (i.e., the “rules”) that must be followed to access the services provided by the program.
* When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a Service-Oriented Architecture or SOA.
* A SOA approach is one where our overall application makes use of the services of other applications.
* A non-SOA approach is where the application is a single standalone application which contains all of the code necessary to implement the application.

**A Service-Oriented Architecture has many advantages**

* we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit)
* the owners of the data can set the rules about the use of their data.



#### Google geocoding web service

* Geocoding is the process of converting addresses into geographic coordinates which you can use to place markers on a map or position the map.
* The following is a simple application to prompt the user for a search string, call the Google geocoding API, and extract information from the returned JSON.

import urllib.request, urllib.parse, urllib.error

import json

serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

while True:

address = input('Enter location: ')

if len(address) < 1:

break

url = serviceurl + urllib.parse.urlencode( {'address': address})

print('Retrieving', url)

uh = urllib.request.urlopen(url)

data = uh.read().decode()

print('Retrieved', len(data), 'characters')

try:

js = json.loads(data)

except:

js = None

if not js or 'status' not in js or js['status'] != 'OK':

print('==== Failure To Retrieve ====')

print(data)

continue

print(json.dumps(js, indent=4))

location = js['results'][0]['formatted\_address']

print(location)

#### Spidering Twitter using Twitter API

Social media provide a number of web services in the form of API’s which can be made use of

in order to analyse text content available on social media to perform aplications such as opinion

mining and sentiment analysis.

**Twitter authentication**

* A twitter account holder should create an application through the account,by which each user

is given the following keys and tokens for authentication

Consumer Key

Consumer Token

Token key

Token Secret

* These keys/tokens are updated in the hidden.py file to do twitter spidering.

def oauth():

return {"consumer\_key": "h7Lu...Ng",

"consumer\_secret": "dNKenAC3New...mmn7Q",

"token\_key": "10185562-eibxCp9n2...P4GEQQOSGI",

"token\_secret": "H0ycCFemmC4wyf1...qoIpBo"}

**Program**

import twurl

import ssl

# https://apps.twitter.com/ # Create App and get the four strings, put them in hidden.py

TWITTER\_URL = 'https://api.twitter.com/1.1/statuses/user\_timeline.json'

# Ignore SSL certificate errors

ctx = ssl.create\_default\_context()

ctx.check\_hostname = False

ctx.verify\_mode = ssl.CERT\_NONE

while True:

print('')

acct = input('Enter Twitter Account:')

if (len(acct) < 1):

break

url = twurl.augment(TWITTER\_URL, {'screen\_name': acct, 'count': '2'})

print('Retrieving', url)

connection = urllib.request.urlopen(url, context=ctx)

data=connection.read().decode()

print(data[:250])

headers = dict(connection.getheaders())

# print headers

print('Remaining', headers['x-rate-limit-remaining'])

## Using Databases and SQL

#### What is a DB?

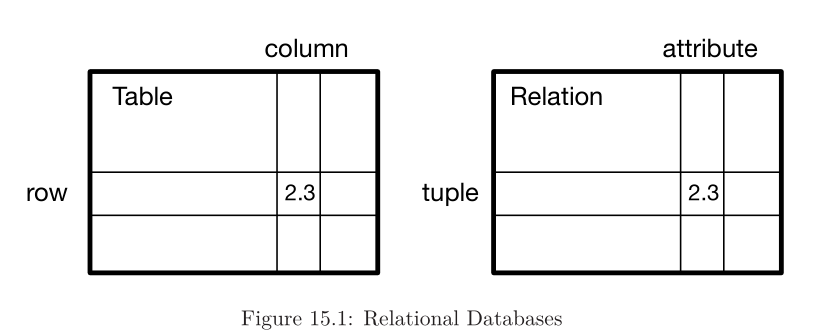
* A database is a file that is organized for storing data
* database is on disk (or other permanent storage), so it persists after the program ends.
* Because a database is stored on permanent storage, it can store far more data than a dictionary
* database software is designed to keep the inserting and accessing of data very fast, even for large amounts of data
* Examples: Oracle, MySQL, Microsoft SQL Server, PostgreSQL, and SQLite

#### SQLite

* SQLite is a very common database and is already built into Python
* SQLite is designed to be embedded into other applications to provide database support within the application

#### Database Concepts

* The primary data structures in a database are: tables, rows, and columns
* They are more formally referred to as relation, tuple, and attribute



#### SQLite Browser

* Many operations can be done more conveniently using software called the Database Browser for SQLite.
* Using the browser you can easily create tables, insert data, edit data, or run simple SQL queries on the data in the database.

#### Creating Tables

* Databases require more defined structure than Python lists or dictionaries
* When we create a database table we must tell the database in advance the names of each of the columns in the table and the type of data which we are planning to store in each column

import sqlite3

conn = sqlite3.connect('music.sqlite')

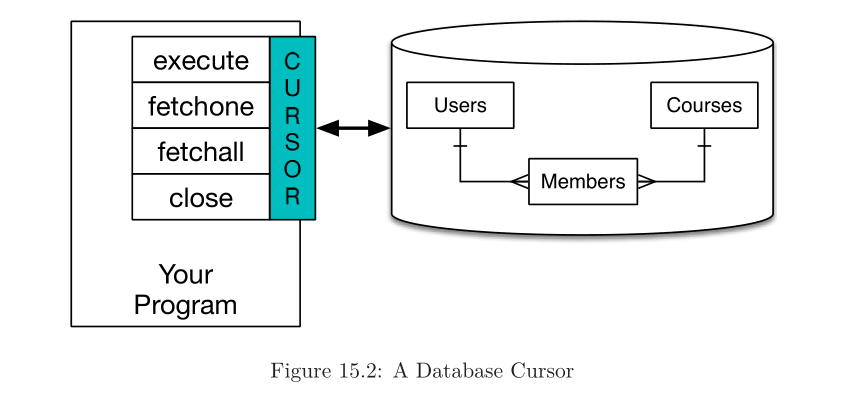
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')

cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()

* The connect operation makes a “connection” to the database stored in the filemusic.sqlite3 in the current directory.
* If the file does not exist, it will be created.
* A cursor is like a file handle that we can use to perform operations on the data stored in the database.
* Calling cursor() is very similar conceptually to calling open() when dealing with text files.
* Once we have the cursor, we can begin to execute commands on the contents of the database using the execute() method.



#### Insertion and Deletion

import sqlite3

conn = sqlite3.connect('music.sqlite')

cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',

('Thunderstruck', 20))

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',

('My Way', 15))

conn.commit()

print('Tracks:')

cur.execute('SELECT title, plays FROM Tracks')

for row in cur:

print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')

conn.commit()

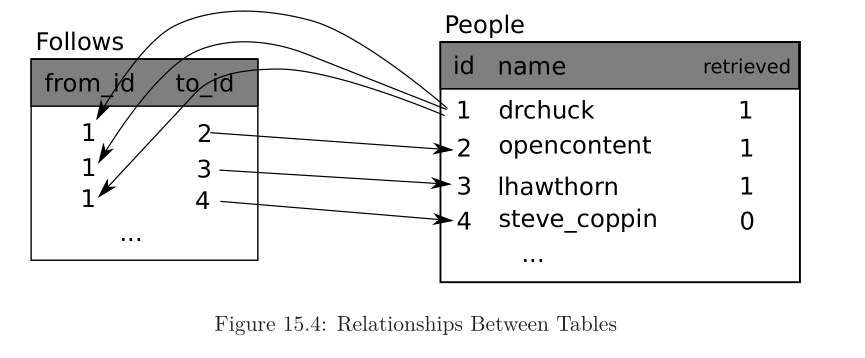
cur.close()

* We specify the values as question marks (?, ?) to indicate that the actual values are passed in as a tuple ( 'My Way',15 ) as the second parameter to the execute() call.

#### Basic Data Modeling

* We create multiple tables and make links between those tables.
* The act of deciding how to break up your application data into multiple tables and establishing the relationships between the tables is called data modeling.

#### Programming with Multiple Tables



* The basic patterns are:
* 1. Create tables with primary keys and constraints.
* 2. When we have a logical key for a person (i.e., account name) and we need the id value for the person, depending on whether or not the person is already in the People table we either need to: (1) look up the person in the People table and retrieve the id value for the person or (2) add the person to the People table and get the id value for the newly added row.
* 3. Insert the row that captures the “follows” relationship.

#### Constraints in database tables

* we can tell the database system that we would like it to enforce a few rule
* cur.execute('''CREATE TABLE IF NOT EXISTS People (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)'‘’)
* cur.execute('''INSERT OR IGNORE INTO People (name, retrieved) VALUES ( ?, 0)''', ( friend, ) )

#### Retrieving and inserting records

* If the account exists, we must look up its id value. If the account does not yet exist in the People table, we must insert the record and get the id value from the inserted row.

friend = u['screen\_name']

cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',

(friend, ) )

#### Three Kinds of Keys

* A logical key is a key that the “real world” might use to look up a row. In our example data model, the name field is a logical key.
* A primary key is usually a number that is assigned automatically by the database. It uniquely identifies a record. Ex: id
* A foreign key is usually a number that points to the primary key of an associated row in a different table.

#### Using JOIN to retrieve data

* SQL uses the JOIN clause to reconnect normalized tables. In the JOIN clause you specify the fields that are used to reconnect the rows between the tables.
* SELECT \* FROM Follows JOIN People ON Follows.from\_id = People.id WHERE People.id = 1

import sqlite3

conn = sqlite3.connect('friends.sqlite')

cur = conn.cursor()

cur.execute('SELECT \* FROM People')

count = 0

print('People:')

for row in cur:

if count < 5: print(row)

count = count + 1

print(count, 'rows.')

cur.execute('SELECT \* FROM Follows')

count = 0

print('Follows:')

for row in cur:

if count < 5: print(row)

count = count + 1

print(count, 'rows.')

cur.execute('''SELECT \* FROM Follows JOIN People

ON Follows.to\_id = People.id

WHERE Follows.from\_id = 2''')

count = 0

print('Connections for id=2:')

for row in cur:

if count < 5: print(row)

count = count + 1

print(count, 'rows.')

cur.close()