Welcome to an introduction to Python, Part 2

G. Benoît

 We've covered the absolute basics. Now let's move into useful python activities.

For loops and Break

Remember the idea of data structures. A list is a set of objects (like Strings, integers, whatever) contained in a single "data structure" called a "list". Lists are indicated by the use of [ ] in the variable declaration:

Looping or Iterating through a list

```
names = ["Bix", "Suky", "BabyKitty"]    # these are my cats
for name in names:
        print(name)
```

The "`names`" (plural) is a variable holding three strings, all in a single list (called "`names`"). The for loop *iterates* through the list. Iteration means the temporary variable (called "`name`" in the singular) is assigned the value from "`names`"; the name is printed. Next we circle back and get the next element in the list and store that value in name and print it, too.

When the `for` statement starts, the first value in `names` is "`Bix`". The print statement then prints "Bix"; since we have more elements in the names list, let's keep going. Now `name` is = to "Suky". Are there more elements in the list? Yup! So let's keep going ... `name` now = "BabyKitty". We print that name, too. Any more in the list? Nope. So the for loop automatically stops.

Python makes a big use of these "temporary variables" that are declared in this way.

Iterating through a letters in a String

Since Strings are really a numbered sequence of letters, we can identify each one individually. If we have a string we can print each letter individually by using this for loop syntax:

```
for x in "paperwork":           # each letter in "paperwork" is stored in "x" and then printed.
        print(name)
```

Nested Loops

We can put a **loop inside another loop**. Say we want to cycle through two lists at the same time. The first list contains first name, the second contains last names:

```
first_names = ["Ming", "Alysson", "Sophia", "Evan", "Anthony", "Kim"]
last_names = ["Wa", "Jones", "Demitrious", "Jeffrey", "Jones", "Kim"]

for x in first_names:
        for y in last_names:
                print(first_names, last_names)
```

Break

Breaking out of a loop

At times we need to exit a loop when some condition is met. For instance say we have a list of names and we print them individually. If we want to locate a certain name, say "Lars", we want to stop processing the data because we've found what we want. Here we **break** out of the loop early:

```
names = ["Ming", "Tom", "José", "Lars", "Larry"]
for x in names:
        print(x)        # each name will be printed as we move thru the list
        if x == "Lars":
                break
```

More about functions

Functions are pretty flexible in python. Some really handy features include using a **default** parameter. Using a default value means we don't have to check for an empty or null value. For instance, if we have a function that accepts a person's name. But what if when we call that function we don't have the person's name? By using a default value we're assured that the variable will have some value.

```
def welcome(name = "Guest"):
        print("Welcome, " + name)

welcome("Tom")
welcome("明娃")
welcome()
```

Pretty cool. Now say we have a bunch of people coming to a library presentation. The names may be stored in a list that we can iterate thru, so functions can accept **lists as parameters**:

```
guests = ["Tom", "Jane", "Maria", "Theo", "Ida"]        # our list of guests to our presentation

def welcome(guests):                                    # define a function that accepts a list
    for x in guests:                                    # iterate thru the guest list
        print("Welcome, " + name)                       # print out their welcome

welcome(guests)                                         # now call the function
```

Return statement: Functions usually do some work and determine an answer and send the answer back to whatever code calls the function. Here are two small examples, one with a number and other with a string:

```
def paycheck( hours_worked, rate_of_pay ):
        return hours_worked * rate_of_pay

print(paycheck(35, 15.00))      # notice we can pass the function to print

def weekly_payroll(last_name, first_name):
        print("Welcome, ", first_name, last_name)
        print("Your weekly paycheck is ", paycheck(35,10))
```

Finally, since functions work with lists, why not other iterators, such as **dictionaries**? Notice, too, that the order of the parameters doesn't matter. This is called the key=value syntax.

```
def print_dept_staff(staff2, staff1, staff3):
        print("The dept manager is ", staff1)
        print("The programmer is ", staff3)
        print("The web designer is ", staff2)

print_dept_staff(staff1 = "Jane", staff2 = "Tom", staff3 = "Tony")
```

File Reading and Writing

Reading and writing data is absolutely critical. Recall the different types of stored data we have (files, databases, etc). There are different ways, then, of accessing these data files. Let's start with the basics.

All files and directories (folders) have access "modes". For our files, we're interested in reading data, appending data to an existing file, writing data, and creating a file if it doesn't exist. This is part of "file handling."

- "r" = read; the default value when trying to read a file. If the file doesn't exist, tho, we'll get an error message.
- "a" = append; opens a file for appending new data and creates a file if it doesn't already exist.
- "w" = write; opens a file for writing - note that we'll get an error if the file doesn't exist.
- "x" = create; create the file for us, but we get an error if the file *does* exist. [to prevent accidentally clobbering the file]

In Python, files are handled as "binary" [for pictures, or binary data] or "text." Text is the default. So let's create a text file - just use your text editor and write up some text; or cut-and-paste text from say Wikipedia. In my example below I'll save the text data in a file called "`yourfile.txt`." [After you've practiced with a text file, try opening an.xml, .json, .yaml, and .csv or even a Microsoft Word document and see what happens.]

It's usually best to check whether or not the file exists before trying to read/write to it. Since "r" and "t" are the default, we don't have to use them. I'll use them here for demonstration, tho.

```
myfile = open("yourfile.txt", "rt")
```

This is the most basic way of reading a file. The variable "myfile" is called the "file handle." We issue the command to "open" the file - that just allows us to do something to it. We have to specify what, such as "reading" the data:

```
myfile = open("yourfile.txt", "rt")
print(myfile.read())                    # to print all the file contents on the screen
```

Readings parts of a file: number of characters or by line

```
myfile = open("yourfile.txt", "rt")
print(myfile.read(10))                  # to read the first 10 letters
```

Reading a single line:

```
print(myfile.readline())
```

But it makes more sense to iterate through the file so we can access all the lines. It's not an uncommon practice to have to read a file line by line as we parse the file or check something about the file before continuing to process.

```
myfile = open("yourfile.txt", "rt")
for each_line in myfile:
        print(each_line)
```

Close the file!

Even tho some commands allow automatic closing of files, we always want to close our files by issue the command close(), e.g., `myfile.close()`

Put it all together for our first little demo. Notice I'm using the ''' kind of comments. Get in the practice of using them and we'll apply them shortly:

```
''' This is a demo python script for reading a file '''

fileToRead = "yourfile.txt"

print("-"*60)
print("Reading the entire contents:")
myfile = open(fileToRead, "rt")                # notice we're using a variable to hold the file name
print(myfile.read())
myfile.close()

''' part 2 reading lines - let's add output line numbers, too '''
print("-"*60)
line_count = 1                                 # let's create a variable to print line numbers

myfile = open(fileToRead, "rt")
for each_line in myfile:
        print(line_count, ": ", each_line)     # look at the syntax very carefully
myfile.close()
```

Most of the time we have files to which we want to add data. You can imagine creating a digital collection. You want to extract data from various resources - the library collection, archives, maybe some images, perhaps content/text of an expert's book - and shape all these resources into a single document. This integration of heterogeneous data is typical in web page design and in exporting data to .xml and .json file formats. Before tackling that we appending data to an existing file (or automatically creating a new file and then adding text) and other commands for deleting and creating files.

Append/Auto create; New File; Overwrite a file

Does the file exist? Recall have some options:

"x" - Create - will create a file, returns an error if the file exist

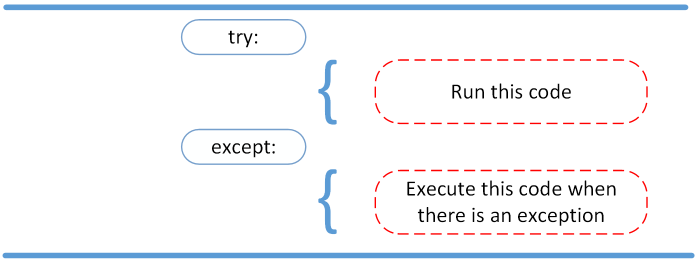
"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

```
‘’‘ in this demo .py file, we need to import the os module (the python library that allows our scripts
    to communicate with the operating system and its functions ’‘’

import os

‘’‘ Append ’‘’
f = open("mytestfile.txt", "a")
f.write("This is the new content to be added to the file.")
f.close()

‘’‘ read the file ’‘’
# open and read the file after the appending:
f = open("mytestfile.txt", "r")
print(f.read())

‘’‘ open a new file if it doesn’t exist and write the data.
If the file already exists, it may be overwritten. ‘’‘

f = open("another_test_file.txt", "w")
f.write("Yikes!  The file will be clobbered.")
f.close()

# open and read the file after the appending:
f = open("another_test_file.txt", "r")
print(f.read())

‘’‘ let’s remove the demo file ‘’‘
if os.path.exists("final_test_file.txt"):       # important! Note here we demo checking if the file exists.
    os.remove("final_test_file.txt")
else:
    print("The file does not exist.")
```

Exceptions - error checking

Any time we read or write data to/from a file or to any data source, we should “wrap it” in an **exception block.** In python the exception block is called a **try/except**. This command tells the script to “do all the commands in the ”try” area of the block; if you can’t, then do none of them and let the user know there’s a problem (called an “exception”). Some people say “try and catch.”



Here's the basic structure, using a file.log. It is very useful and common in real practice to show only the most common errors to the end-user on screen (because there are *thousands* of exceptions), to show a generic error if one of the non-common errors occurs, and to save the errors in an error log. By analyzing the logs (called *transaction log analysis*) we can determine end-user behaviors and help to debug our code. Using our file.log idea ...

```
file_to_read = 'file.log'

try:
    with open(file_to_read) as file:
        read_data = file.read()
except:
    print('Could not open file.log')
```

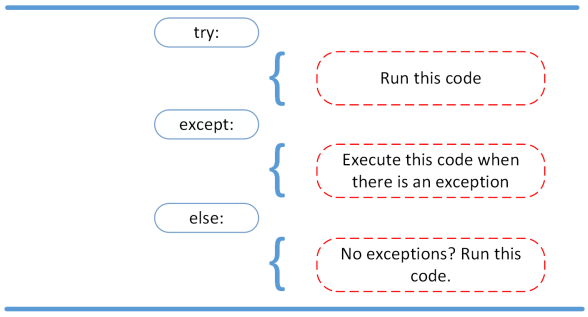
Our error message isn't that interesting. Let's change it a little:

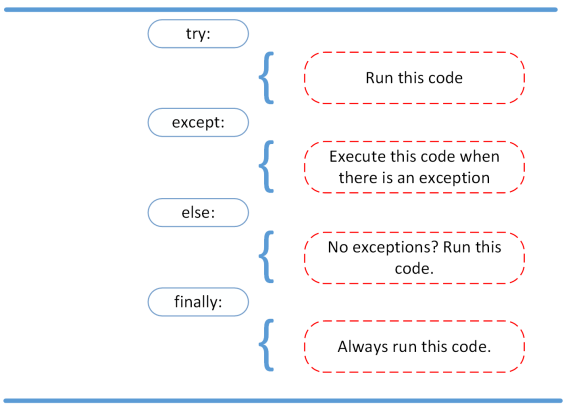
```
try:
    with open(file_to_read) as file:
        read_data = file.read()
except:
    print("I'm sorry, I couldn't read the file: ")
    print("Lo sentimos, el archivo no se puede leer: ")
    print("Xin lỗi, tập tin không thể đọc được: ")
    print("抱歉，无法读取该文件: ")
    print(file_to_read)
```

Here's an example of the common errors: **File not found.** It's so common, there's actually a built-in error type!

```
try:
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

Notice how try/except will give more power by adding an "else" statement. And then there's always the "finally" - code that you might want to run no matter what. For instance, you might have a generic copyright statement or contact data to show.





You can write your own exceptions. These are called "assertions." Please check the class texts for specifics. In addition, always check the official documentation for any software you employ. Check out https://docs.python.org/3/library/exceptions.html

---

This example from the Net shows the programmer trying to test whether the code will run on Linux. If the code cannot run on Linux, then there will be an error message. Else if the code *can* run on Linux, then the script will try to read the file. And, of course, if the file is not found, then we can't read it so show a message. Finally let's clean up no matter what.

```
try:                                        # start the try/except block
    linux_interaction()                     # test if True for Linux
except AssertionError as error:             # if not true, show the actual error*
    print(error)                            # print(error)
else:                                       # otherwise …
    try:                                    # let's try to open a file
        with open('file.log') as file:
            read_data = file.read()         # if so, read the data
    except FileNotFoundError as fnf_error:  # yikes, we can't read the file
        print(fnf_error)                    # if there's an error print the error
finally:
    print('Cleanup - thanks for visiting.')
```

* notice that errors can be saved in a temporary variable for us to use. Here the error message is stored in a variable called "error".

JSON file and Python

It's extremely to read/write data to/from python scripts and json file. JSON files look a lot like python dictionaries so let's practice reading and writing them.

```
import json                             # first call the module that lets us use json

# some demo, hard coded data.  imagine, tho, that the data came from a rdbms or other source
x =  '{ "fullname": "John Smith", "age": 23, "city": "San Jose" }'

# let's take the contents of x… in "y"
```

```
y = json.loads(x)

# the result is a Python dictionary:
print(y["fullname"])                    # notice we can extract the value of data by the "name"
print(y["age"])                         # gives us lots of control over our data and clarity in code
```

Example

Here's let's say we have data (yeah) and we want to sort the data and export them into a .json file. [Then we can share the data with others - and extract just what we need for various projects.] NOTE the following: let's keep the indentation as 4 space and notice the different separators (remember we have to clean up the data a bit. You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values.

For a "real-world" example of json - check out this Magic Realism entry at LCSH (http://id.loc.gov/authorities/subjects/sh85079627.html):

```
'''
In this example, we'll imagine we've imported data from different sources and now want to sort them and save
them for a retrieval project.
'''

import json

x = {
  "name": "Abate, Sandro",
  "title": "Learning from Magic Realism",
  "onloan": True,
  "digital": False,
  "main_characters": ("Luis", "Maria"),
  "pages": 244,
  "subjects": [
    {"LCSH": "Magic Realism", "LCCN": PN56.M24 M335 2014},
    {"Other Cite": "LC Control Number", "ID": 2014947173}
  ]
}

# sort the result alphabetically by keys:
print(json.dumps(x, indent=4, sort_keys=True))
```

Outputting data to a .json file

When python saves a file, the file needs to be "serialized." Serialization (from https://docs.python-guide.org/scenarios/serialization/) defines it: *Data serialization is the process of converting structured data to a format that allows sharing or storage of the data in a form that allows recovery of its original structure. In some cases, the secondary intention of data serialization is to minimize the data's size which then reduces disk space or bandwidth requirements."*

```
import json

x = {
  "name": "Abate, Sandro",
  "title": "Learning from Magic Realism",
```

```
  "onloan": True,
  "digital": False,
  "main_characters": ("Luis", "Maria"),
  "pages": 244,
  "subjects": [
    {"LCSH": "Magic Realism", "LCCN": PN56.M24 M335 2014},
    {"Other Cite": "LC Control Number", "ID": 2014947173}
  ]
}

# here's the main differences: from printing to the screen print(json.dumps(x, indent=4, sort_keys=True))
# we'll now print to a file:

with open("my_first_json_file.json", "w") as data_to_save:
    json.dump(x, data_to_save)
```

Reading in the serialized .json file:

```
with open("my_first_json_file.json", "r") as data_to_read:
    data = json.load(data_to_read)
```

A full example of "serializing data" and reading the data back in...

```
import json

x = {
  "name": "Abate, Sandro",
  "title": "Learning from Magic Realism",
  "onloan": True,
  "digital": False,
  "main_characters": ("Luis", "Maria"),
  "pages": 244,
  "subjects": [
    {"LCSH": "Magic Realism", "LCCN": "PN56.M24 M335 2014"},
    {"Other Cite": "LC Control Number", "ID": 2014947173}
  ]
}

''' here's the main differences:
    from printing to the screen
    print(json.dumps(x, indent=4, sort_keys=True)) -
       we'll now print to a file '''

with open("my_first_json_file.json", "w") as data_to_save:
    json.dump(x, data_to_save)

# Reading in the serialized .json file:

with open("my_first_json_file.json", "r") as data_to_read:
    data = json.load(data_to_read)
    print(data)            # print all the data like a string

# we must convert the "serialized data" into a string
# the "with open()" command automatically closes the file, so
# we must open it again:
with open("my_first_json_file.json", "r") as data_to_read:
```

```
        json_string = json.load(data_to_read)

# the result is a Python dictionary and we can access data by "key":
print(json_string["name"])

# can you wrap these data in a web page?!
```

A Challenge:

Take the above code and (a) add some try/exception for the file reading/writing and (b) see if you can output the data in a web page, by integrating HTML tags (as strings). You can do it!

Advanced fyi:

File formats for sharing data evolve and the programming languages using those files are moving to be similar in syntax to make it easier to ingest (read in) data and output those data. Here are some python examples:

CSV file (flat data)¶

The CSV module in Python implements classes to read and write tabular data in CSV format.
Simple example for reading:

```
# Reading CSV content from a file
import csv
with open('/tmp/file.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Simple example for writing:

```
# Writing CSV content to a file
import csv
with open('/temp/file.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(iterable)
```

The module's contents, functions, and examples can be found in the Python documentation.

YAML (nested data)¶

There are many third party modules to parse and read/write YAML file structures in Python. One such example is below.

```
# Reading YAML content from a file using the load method
```

```
import yaml
with open('/tmp/file.yaml', 'r', newline='') as f:
    try:
        print(yaml.load(f))
    except yaml.YAMLError as ymlexcp:
        print(ymlexcp)
```

Documentation on the third party module can be found in the PyYAML Documentation.

JSON file (nested data)¶

Python's JSON module can be used to read and write JSON files. Example code is below.
Reading:

```
# Reading JSON content from a file
import json
with open('/tmp/file.json', 'r') as f:
    data = json.load(f)
```

Writing:

```
# Writing JSON content to a file using the dump method
import json
with open('/tmp/file.json', 'w') as f:
    json.dump(data, f, sort_keys=True)
```

XML (nested data)¶

XML parsing in Python is possible using the xml package.
Example:

```
# reading XML content from a file
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

More documentation on using the xml.dom and xml.sax packages can be found in the Python XML library documentation.

End of the lesson.

Additional Notes - just fyi:

I'm sharing a note I prepared for myself a while ago. It's offered to give you a challenge reading code you've never seen before and *to suggest there's lots of other data types (bytes, sql, etc.), that we may have to use. Each of these has specific needs, too. SQL, for example, we must create a "connection" between our program and the database source. [And it's common* not *to have permissions so we have to address that, too!] With your skill set so far you should be able to "scrape" webpages! Check out the https://scrapy.org homepage.*

We know we can read/write text and binary files ... how about interacting with blobs (binary large objects) and MySQL? Pretty common technique for some media files. [To be sure you could use Java Media library or Python's media libraries, too, to write movies, pdfs, to/from python alone or with sql.]

You may want to use python to save media files … convert to base64:

```
import base64

video_stream = "hello"

with open('file.webm', 'wb') as f_vid:
    f_vid.write(base64.b64encode(video_stream))

with open('file.webm', 'rb') as f_vid:
    video_stream = base64.b64decode(f_vid.read())

print video_stream
```

Python & MySQL

Note that the below hardcodes the connection data - better to use a config file to store and retrieve your own settings.

```
import mysql.connector
from mysql.connector import Error
from mysql.connector import errorcode

def convertToBinaryData(filename):
    #Convert digital data to binary format
    with open(filename, 'rb') as file:
        binaryData = file.read()
    return binaryData

def insertBLOB(emp_id, name, photo, biodataFile):
    print("Inserting BLOB into python_employee table")

    try:
        connection = mysql.connector.connect(host='localhost',
                                database='python_db',
                                user='Bears',
                                password='goBears')

        cursor = connection.cursor(prepared=True)

        sql_insert_blob_query = """ INSERT INTO `python_employee`
                          (`id`, `name`, `photo`, `biodata`) VALUES (%s,%s,%s,%s)"""

        empPicture = convertToBinaryData(photo)
        file = convertToBinaryData(biodataFile)

        # Convert data into tuple format
```

```
        insert_blob_tuple = (emp_id, name, empPicture, file)

        result  = cursor.execute(sql_insert_blob_query, insert_blob_tuple)
        connection.commit()
        print ("Image and file inserted successfully as a BLOB", result)

    except mysql.connector.Error as error :
        connection.rollback()
        print("Failed inserting BLOB data into MySQL table {}".format(error))

    finally:
        #closing database connection.
        if(connection.is_connected()):
            cursor.close()
            connection.close()
            print("MySQL connection is closed")
```

E.g.,

```
insertBLOB(1, "Eric", "D:\images\richard_photo.png", "D:\images\richard_bioData.txt")
```

Python & MySQL

iPads and mobile devices use SQLite:

```
import sqlite3
conn = sqlite3.connect('database.db')
cursor = conn.cursor()

with open("...", "rb") as input_file:
    ablob = input_file.read()
    cursor.execute("INSERT INTO notes (id, file) VALUES(0, ?)", [sqlite3.Binary(ablob)])
    conn.commit()

with open("Output.bin", "wb") as output_file:
    cursor.execute("SELECT file FROM notes WHERE id = 0")
    ablob = cursor.fetchone()
    output_file.write(ablob[0])

cursor.close()
conn.close()
```

Google Cloud (never used this)

https://google-cloud-python.readthedocs.io/en/0.32.0/storage/blobs.html