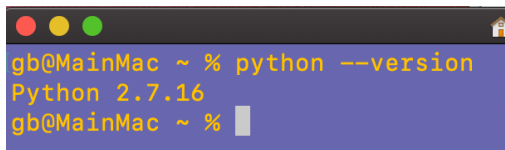


Review of control-of-flow for week 2

GB - This summary document reviews some of the contents of week 2 along with some hints at applications of control-of-flow statements. This document introduces, too, file read/write, try/except blocks, error checking, and how to connect to an SQL server and use several file types. 1/11/21, 3:00 PM

Downloading and installing python

1. First check whether python is already installed on your computer. On Mac/Unix/Linux OS computers, python is already installed. Access the command-line/terminal window:
 - a. Linux: Ctrl-Alt-T, Ctrl-Alt-F2
 - b. Windows: Win+R > type `powershell` > Enter/OK
 - i. You may want to read more about Python on Windows on [Python's documentation site](#).
 - c. MacOS: Finder > Applications > Utilities > Terminal
2. Once the terminal window is ready, issue the command to check: `python --version`



```
gb@MainMac ~ % python --version
Python 2.7.16
gb@MainMac ~ %
```

3. There may be multiple versions on your machine: check by typing `python --version` (for versions 2.7.x) and `python3 --version` (for versions 3.x). *Note: depending on your computer setup you may need to replace the command “python” with “python3” throughout the rest of these readings and exercises.*

As a fun little test, let's start a text editor application, such as [BBEdit](#), [Atom](#), NotePad++, and write our first little script. This script you'll run from the terminal window!

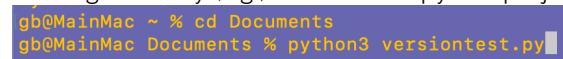
Your first script - test whether python is installed:

Wait! Before you create the file, remember that python relies on *indentation* to identify “code blocks.” The indentation must be *either* four spaces *or* a tab; never both in the same file. So in the demo below, the first “print” line is intended by tapping the space bar four times.

```
import sys

if not sys.version_info.major == 3 and sys.version_info.minor >= 6:
    print("Python 3.6 or higher is required.")
    print("You are using Python {}.{}.".format(sys.version_info.major, sys.version_info.minor))
    sys.exit(1)
```

When you're done save this file as “versiontest.py” [notice there are *no* spaces in the file name]. Make sure your current working directory (e.g., “Documents/pythonprojects/”) is the same that holds your new .py file.



```
gb@MainMac ~ % cd Documents
gb@MainMac Documents % python3 versiontest.py
```

All computer programming and scripting languages rely on “control of flow” statements, functions and methods, data structures, file reading/writing, and error checking. Of course there is a lot more and each language has its own syntax but the concepts remain strong across all languages. We'll explore each of these now.

Using Python's Command Line

Keeping to our terminal window we'll start the actual python program (or "interpreter") that will interpret our commands and respond immediately. To **exit** python, enter **exit()** and press the return key.

1) Start python by typing **python** and pressing the enter or return key.

```
gb@MainMac Documents % python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

2) Notice python has its own prompt: the three >>>

Functions and methods are pre-written commands that we use as the basics of the language. Functions are independent, defined blocks of code that perform some task. Functions may accept "arguments" (or parameters), data that we supply for the function to do its job. Functions may "return" a value, the result of the function's work. In this example the `print()` function takes an argument "Hello!", and "prints" the results on the stdout (the monitor).

```
>>> print("Hello!")
Hello!
>>> x = "Hi, again!"
>>> print(x)
Hi, again!
>>> 
```

We can also **declare variables** and use them. A **variable** is a container with a name that holds a value. E.g., "x" is a variable name as could be `welcome_greeting`. Variables cannot begin with certain symbols (such as period, colon, exclamation point) and cannot contain spaces. Once a variable is "declared" and assigned a value "instantiated" we can use it in code. Here we see "`print()`" function being passed a parameter ("Hello"); next we see a variable declared (x) and passed as a parameter, too. And the value of the parameter is used in the output.

3) **Control-of-Flow and Loops; Variables and more:**

Demo Table 1	
#comments	It's best to include comments in the code so you'll remember what you're doing and others can debug the code. There are 2 kinds: the # and a comment block using <code>"""</code>
if	<pre># an if statement if 5 > 3: print("five is indeed bigger than 3") # an if statement using variables x = 5 y = 3 if x > y: print("and it works with variables")</pre>
if ... elif	<pre># an if-elif statement using variables x = 5 y = 3 if x > y: print("Five is indeed bigger!") elif x == y: print("well, x and y are equal! Weird.")</pre>

if ... else	<pre># an if-else statement using variables x = "Tom" y = "Turkey" if x == y: print("The two variables are equal.") else: print("No, the two variables are not the same.")</pre>
Shortcuts	<pre>x = 82 y = 92 if y > x: print("You scored and A") """ in the above statement notice the syntax. if the value of y is greater than that of x, then print "You score and A." """ print("Grade A") if y > x else print("B") """ notice the syntax about the outcome (the print) appears first, then the comparison - the if statement followed by the else. Get fa- miliar with this notation because it's used in "lambda functions". """</pre>
Combine comparisons with and	<pre>a = 5 b = 10 c = 15 if a > b and a > c: print("both statements are true.") if a > b or a > c: print("at least one of these statements is true.")</pre>
Nested if statements	<p>Sometimes we want to check a condition and see if it is true. If that condition is true, then we might want to do tests on other conditions. For example, "if the student is in class 100" we want to assign grades.</p> <pre>student_in_class = True score = 93 if student_in_class: # same as if student_in_class == True: if score > 90: print("Grade A") else if score > 80 and score <= 89: print("Grade B") else if score > 70 and score <= 70: print("Grade C") elif: print("Grade D, sorry.")</pre>
Pass statement	<p>There are times when you're coding and you kind of know what you want to include but you're not finished. The pass statement allows us to use the code even if we're not ready to finish the code and not get an error.</p> <pre>student_Name = "Tom" is_enrolled = True if is_enrolled: pass</pre>

# types of variables built-into python	<p>Depending on the kind of data we have we use different kinds of variables to hold those data. Here's the basics.</p> <p>integer counting numbers, e.g., 0,2,3,6,333,142396</p> <p>floating point 3.141458372</p> <p>char 'a', 'b'</p> <p>string "Hi, folk" can use single quotes, too, 'Hey!'</p> <p>str for text, e.g., x = "Smith"</p> <p>int, float, complex for numbers, e.g., i = 3.1432</p> <p>sequence types list, tuple, range</p> <p>mapping types dict</p> <p>boolean types bool</p> <p>[there are others but we'll not use them now.] Notice that strings are wrapped in quotes. Numbers are not. This is how python determines the data type.</p>
<p>Using variables.</p> <p>Variables can be "global" or "local." A global var can be used anywhere in the code. A local variable is used only in the code block where it is declared.</p>	<pre> """ Notice that the quotes can be double or single when wrapping a string. That's how we can use the ' in the word let's. If the quotes are unbalanced then the code will fail. """ s = "Let's go to the beach!" print("It's such a great day " + s) </pre> <pre> >>> s = "Let's go to the beach!" >>> print("It's such a great day. " + s) It's such a great day. Let's go to the beach! >>> </pre>

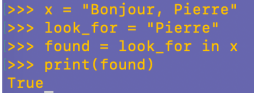
Test yourself on the contents of Demo Table 1. When you understand it all, proceed to the next table(s).

Demo Table 2	<p>Knowing the value of a variable and what parts of the code can change the value is awfully important! Let's practice controlling the "visibility" of the variable by defining some functions and testing. Then we'll "promote" the variable to global as a test.</p>
defining a function	<pre> """ functions require a "keyword", def, to define a function's code block. Then any variable declared in that block is visible only to that block of code. Imagine if you have two variables called x ... """ </pre>

<p>the keyword def for defining functions</p>	<pre> x = "fun" # notice this var is not part of a function; it is global def whatToDo(): x = "Go to the beach" print("What to do? Something like " + x) # to end our function press the return key twice. whatToDo() # notice that functions are defined but not used # until we "call" them into action print("Let's do something " + x) >>> s = "Let's go to the beach!" >>> print("It's such a great day. " + s) It's such a great day. Let's go to the beach! >>> x = "fun" >>> def whatToDo(): ... x = "Go to the beach" ... print("What to do? Something like " + x) ... >>> whatToDo() What to do? Something like Go to the beach >>> print("Let's do something " + x) Let's do something fun >>> </pre>
<p>Data Type samples:</p> <p>(there is also complex, set, frozenset, bytes, bytearray, and memoryview types but we'll not use these in our course.)</p>	<pre> x = "Hello, Dave" string type (str) x = 20 integer (int) x = -39.1 floating point (float) x = ["cat", "dog", "cow"] a list - notice the use of [. Lists are great for using lots of variables. x = ("cat", "dog", "cow") a tuple - notice the () x = range(10) a range, extremely useful x = { "name": "Ricky", "age", "27"} a dictionary (dict) - invaluable especially with .json files x = True boolean (True or False) </pre>
<p>Converting between data types</p>	<p>Say we have a variable "x" that refers to an integer, 10. We want to convert that integer to a floating point value. Use the "constructor": e.g.,</p> <pre> x = 10 print(x) 10 # notice the output x = float(10) print(x) # always good to confirm. Try it. 10.0 </pre>

Practice creating lots of variables and printing them. "Cast" (or convert) the variable between different types and see if works and if not what the error messages look like. It's *vital* that you become accustomed to error message, not stressing over them (grin) and learning how to read the error message for debugging your code.

<p>Demo Table 3</p> <p>“Arrays” - python like most language stores “strings” as an array of unicode characters.</p>	<pre> x = “Buenas días” print(x) print(x[0]) print(x[1]) # this is a string # and the entire sting prints # the [] refers to the “index” or the # location of some value at the # used # the letter “B” prints (location = 0) # returns “u” </pre>
<p>knowing where data are being stored is really important. And there are several ways to accessing the data - by position or “index” is common.</p>	<pre> """ Slicing data - we select the start number, the last number, and the number of characters to take at a time """ print(x[0:5:1]) print(x[0::]) print(x[0:len(x):]) print(len(x)) # prints “Buena” # prints “Buenas días” - since we add no # values for last and number at a time # python assumes we want all the data. # notice len(x) - this means “get the # length of the word and store value here # then complete the splice command. # let’s check ... 11 </pre>

Demo Table 4	
	<p>Strings There is a lot of string processing used in information science and informatics. In this demo table we look at some of the most important. It's always important to get to know your data - make no assumptions - and check. Sometimes even an extra space at the end of the word might wreck our code. And when comparing values (e.g., "Hello" ≠ "HELLO") we might want to convert data to all uppercase or all lowercase and so on. Let's try each of these in your terminal window. Using the same string in all examples: x = "Bonjour, Pierre". We send the output of our string commands to the print method so we can confirm.</p>
upper case lower case strip off whitespaces	<pre>x = "Bonjour, Pierre" print(x.upper()) print(x.lower()) print(x.strip())</pre>
checking strings is a substring found?  <pre>>>> x = "Bonjour, Pierre" >>> look_for = "Pierre" >>> found = look_for in x >>> print(found) True</pre>	<pre>""" sometimes we check whether data are found in a string. If we might want to replace some characters. Or we might want to break up a string into "tokens" based on some delimiter. This is often the first step in data cleaning and parsing text files for full-text retrieval algorithms. """ look_for = "Pierre" # use the "in" or "not in" keywords: found = look_for in x # is "Pierre" found in the string x? print(found) """ now you try - using the "not in" keyword."""</pre>
breaking up strings splitting replacing breaking up strings with the \t tab	<pre>""" often need to break up strings based on some delimiter, such as the tab (or \t) or a comma. """ print(x.split(",")) # returns ['Bonjour', 'Pierre'] """ now you try - using the "not in" keyword. """ s = "Bontour, Pierre" # the "t" is an error; let's fix that. s = s.replace("t", "j") # the first instance of t is now j print(s) """ here the data are separated by an invisible tab (\t). I inserted the \t to demo its location. NOTE: this will not work on the terminal line - but it's something we use all the time IN the .py source file.""" s = "Tom Jones\t50,000\tRoom 32\t333-1234" """ because it is SO common to export data from a spreadsheet into a "tab-delimited file", we need to know how to extract our data """ print(s.split("\t")) # what do you see?</pre>

<p>About “escape characters” - tab, newline, carriage return</p> <p><code>\t</code></p> <p><code>\n</code></p> <p><code>\r\n</code></p> <p>and apostrophes/quotes</p> <p><code>\'</code> for apostrophe</p> <p><code>\"</code> for quotes</p>	<p>An “escape” character tells the computer to “escape” from its usual processing and treat the following character in a special way. The most common for beginners is “tab”, “new line”, and “carriage return.”</p> <p>Tab is indicated by “<code>\t</code>”</p> <p>New line (as if you pressed the return key) is “<code>\n</code>”</p> <p>Because quotes <i>must</i> be balanced, if we have a string of data wrapped in “ ” and then a “ inside the string, we must “escape” the quote: e.g.,</p> <pre>s = "We've won the "Nobel Prize" for physics!" must become s = "We've won the \"Nobel Prize\" for physics!"</pre> <p>For WINDOWS only there's often a hassle because Windows used to use “<code>\r\n</code>” (carriage return + new line) and that causes a problem interpreting the combination. Programming languages usually will convert the “<code>\r\n</code>” to just “<code>\n</code>”. BUT you cannot be sure about this when reading a data file! Just fyi.</p>
--	--

Practice creating strings and manipulating them.

Demo Table 5	Booleans
With booleans we can test the data, a function, and write code using our control-of-flow statements.	<pre>x = 10 print(x == 10) # the == means “is equivalent to” # so read the statement as “print the result # true if x is equivalent to 10, print false” """ try using the other operands, too: > == < >= <= """</pre>
Using in code	<pre>x = 1000 y = 1032 if x < y: print(x, “ is less than ”, y) else: print(y, “ is less than ”, x)</pre> <p>""" Notice the print statement. Python let's us send a many arguments, separated by a comma, and the string appears correct on screen """</p>

<p>Using in a function as a return value</p> <p>Here, notice at the bottom of the code snippet, x and y are assigned values. Then we pass those values to the function which_is_greater. That function springs into action and sends back to the part of the code that called it the results - the "s"</p>	<p>When we reuse the same calculations or commands a lot it is useful to store them in a function. And since functions can "return" the result of its work and store that result in a variable, it's extremely useful to combine functions and booleans. Here say we're often asking if two variables are greater or less than each other. So let's write a function:</p> <pre>def which_is_greater(x, y): if x < y: s = print(x, " is less than ", y) else: s = print(y, " is less than ", x) return s print("lets compare two variables") x = 300 y = 400 which_is_greater(x, y)</pre>
---	--

So far we have the basics; now let's build upon this foundation.

Demo Table 6	for ... and while loops
<p>for ...</p> <p>while loops</p>	<p>Usually we want to keep something running until something causes it to stop. For example, when we write a program, we want to keep it running until the end-user chooses to quit. Equally, we might already know when to quit something when we have a limited number of items to check, for example, let's say we have five cards. We could say "while we have cards in our hand, lay them down on the table one by one." Often we have a variable just for counting. So we need to count which card we're using and compare that number to the total number of cards. The first time we lay down a card, the count is 1, and then 2, and so on until we reach the end of the cards.</p> <pre>count = 1 number_of_cards = 5 while count < number_of_cards: print("Card # ", i) count += 1 # if we don't increment the count, the loop continues forever</pre>
<p>break statement stops the loop even if the while condition is true</p>	<pre>i = 1 while i < 6: print(i) if i == 3: break i += 1</pre>
<p>continue continues the next iteration, in this case, even if i is 3.</p>	<pre>i = 0 while i < 6: i += 1 if i == 3: continue print(i)</pre>

while and else together! Prints a message once the condition is false.	<pre> i = 1 while i < 6: print(i) i += 1 else: print("i is no longer less than 6") </pre>
--	--

The fundamentals we've seen and hands-on practice should provide a reasonable basis for benefiting from the online text books. In this example, we want to create a menu for end-users. The program runs in a terminal window until the end-user chooses to exit. There are four options: 1, 2, 3, and q. Let's say 1 = "English", 2 = "Español", 3 = "Français", and 0 = quit.

Demo screen:

```

Hi.  In what language would you like your welcome?

```

```

1 = English
2 = Español
3 = Français
0 = Quit

```

```

Enter your choice:

```

Demo code for the above:

```

print("-" * 60)                                # means print - for 60 times
print("Hi. In what language would you like your welcome?\n")
print("1 = English")
print("2 = Español")
print("3 = Français")
print("0 = Quit")
useroption = (int)input("Enter your choice:")
print("-" * 60)

# now let's check the values of our choices. We accept ONLY 1, 2, 3, and the letter "q"
if (useroption in range(0,4)):                  # if the user choice is legit (0, 1, 2, 3)
    # nested if example
    if useroption == 1:
        print("Welcome to our class")
    else if useroption == 2:
        print("¡Bienvenido a nuestra clase!")
    elif:
        print("Bienvenue à notre cours.")
else:
    print("Sorry, only values 1, 2, 3, and 0 are okay.")

```

The above script will run only one time. Let's keep it running until the end-user chooses to exit by applying the while statement. Notice below the use of (int). This means whatever the end-user enters should be converted to an integer. Notice, tho, if the person enters "a", then it mayn't work.

```
print("-" * 60)                # means print - for 60 times
print("Hi. In what language would you like your welcome?\n")
print("1 = English")
print("2 = Español")
print("3 = Français")
print("0 = Quit")
useroption = (int)input("Enter your choice:")
print("-" * 60)
```

"""

Here we apply a "boolean flag". That means we set up a condition to be True until the end-user enters some data that causes the boolean flag to become False. Our flag here is "is-Done"

"""

```
isDone = False                # keep going until "isDone" is true

while !isDone                # the isDone is "false", using the ! reverse it to true
    # nested if example
    if useroption == 1:
        print("Welcome to our class")
    else if useroption == 2:
        print("¡Bienvenido a nuestra clase!")
    else if useroption == 3:
        print("Bienvenue à notre cours!")
    else if useroption == 0:
        print("Thanks for visiting")
        isDone = True
    elif:
        print("Sorry only 0, 1, 2, 3 values are legit.")
```

Compare the two approaches. Both are fine. In real practice we try to avoid too many nested if statements and combinations of for, while and if all together. The time it takes to process the code and data can increase exponentially. The measurement of coding algorithm design is called Big O. Big O is a measure, you'll recall of the "worst case scenario" of the algorithm. For our purposes, we don't have to worry about it. But should you continue with python and particularly BigData/Data Science, you should read more about this.

Operators	<p>here's a fuller list of the operands:</p> <ul style="list-style-type: none"> + addition e.g., <code>x + y</code> - subtraction <code>x - y</code> * multiplication e.g., <code>5 * 2</code> / division % modulus (the remainder of 2 numbers, e.g., <code>30 % 2 = 0</code>) ** exponent, e.g., <code>5²</code> is written <code>5**2</code>
Assignment operators	<ul style="list-style-type: none"> = <code>x = 5</code> assign a value of 5 to the variable <code>x</code> += increment 1, e.g., <code>x += 3</code> is the same as <code>x = x + 3</code> -= decrement /= divide, e.g., <code>x /= 3</code> is the same as <code>x = x / 3</code>
Comparison operators	<ul style="list-style-type: none"> == equal to, e.g., <code>x == 5</code> (returns true or false) != not equal to <code>x != y</code> > greater than <code>x > y</code>, e.g., <code>5 > 3</code> < less than <code>x < y</code> >= greater than or equal to <= less than or equal to
Logical operators	<ul style="list-style-type: none"> and returns True if both statements are true, if <code>x < 5</code> and <code>y > 3</code> or returns True if <i>one</i> of the statements is true, <code>x < 5</code> or <code>x < 4</code> not reverses the result; false if the result is true, e.g., if not(<code>x < 5</code> and <code>y > 3</code>):
Identity operators	<ul style="list-style-type: none"> is returns true if both variables are the same object <code>x is y</code> is not returns true if both variables are <i>not</i> the same object "same object" means the same memory location
membership operations	<ul style="list-style-type: none"> in returns True if a sequence with the specified value is present e.g., for <code>x in range(0,5)</code>: means if the value of "x" is not between 0 and 5, then the answer is False. not in returns true if a sequence with the specified value is not present in the object

Part 2:

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 append 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())

```



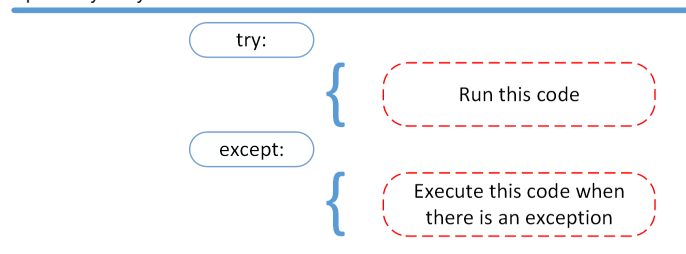
```

''' 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:

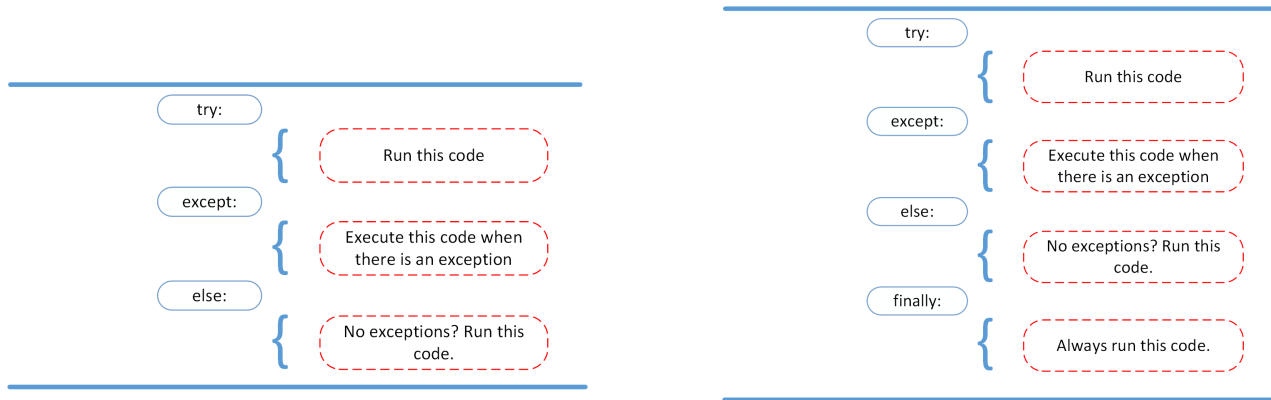
```

```

    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:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleanup - thanks for visiting.')

# start the try/except block
# test if True for Linux
# if not true, show the actual error*
# print(error)
# otherwise ...
# let's try to open a file

# if so, read the data
# yikes, we can't read the file
# if there's an error print the error

```

* 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('/tmp/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 ymlexc:
    print(ymlexc)
```

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>

