Week_02_Sp22 - G Benoit

W200: Some week 2 notes offered as-is. Your instructor may emphasize other aspects of python based on class interests and understanding.

The notebooks start with an agenda or the main points.
Each week we have plans for a small-group session, called a breakout room. We don't always have the time for these sessions; in such cases the answer key would be posted so you can compare your efforts to the answer key.

Here are some main points to consider:

- Remember: there's only one input (**stdin**, standard input), one output (**stdout**, standard output), and error (**stderr**, standard error) active at a time. The default input is the keyboard; output is the screen; error could be screen or to a log file. Our commands change these in a flash, e.g., when we save data to a file we've temporarily set the stdout to a file handle (which in turn communicates with the operating system, asking if we have permission to write the file, and then saving the file. When we close the file handler, default is reset back to stdout. Error messages, we'll see later this term, can be written to a file (and they should be for transaction log analysis) or to the end-user (just the most important ones - they don't need to see weird computer code error messages but we do in development and distributed packages). We'll see python has an class for determining how to react to one of 5 types of error groups.
- Python is a 100% **object-oriented language**; it's just so flexible for coding that we can write "procedural" or "functional" (like any scripting language) or use an entirely object-oriented programming approach. The OOP approach is what we'll work towards for Project 1.
- Note the reliance of **balanced quote marks** (' and ' together or " and "). In most languages the double-quotes are reserved to indicate String; single quotes to indicate a character (single byte).
- Python doesn't use { } for code blocks for tabs *or* 4-spaces. The 4-spaces indentation is the preferred python way (I find it a pain) - if you cut-and-paste code you'll end up w/ a mix of tabs and spaces and you'll get an error. Replace the \n with 4 spaces (global search and replace) and save the file.
- Building up: think of python as having data structures who can be sequenced from most elementary (a bit) through byte through String and int, to float, and then more obviously Object looking ones, like Decimal.
- **Object-Hierarchy**: the built-in objects (like int, String) are "primitives" and we use these to build our own objects. All objects exist in a parent-child relationship. The grandparent of 'em all is the Object itself.
    - Objects we use from python have lots of built-in properties we mayn't know about or use - but they're there - and we see some in the . (dot operator) commands that follow an object or an instance (copy; our variable) of that object. For instance a String is easily *declared* and *instantiated*: x = "hello"

        - Important to note: if a variable is declared (e.g., x) but has no value (= 5), we can ask python if x: [meaning return True if x has a value; return False if it doesn't. This is a very handy shortcut in our coding practices.
    - So notice if x is a String; it inherits all the properties of Python's "String" object - and we can see the *method* that's being inherited because it follows variable, separated by a period: e.g., x.upper() The .upper() means convert to upper-case. The .upper() is a **method** that belongs to String. All objects display "encapsulation", "inheritance", and "polymorphism". More on this later.
- Strings are contiguous blocks of RAM. Unlike C++ where we might have a pointer variable to the first memory location and then dereference that variable to get its value, python does all that work for us.
- **type()** and **id()**. Since we can convert one object into another object type ("cast") - we can (a) to confirm what type of object we have, e.g., type(x). And we can determine the individual ID of each variable using the id().
- The Object Hierarchy of Python:
    - Take a look at this chart. Notice the lines linking an object to its parent object. If a given object doesn't seem to have the method built into it … python goes up the ladder to the parent of that object and looks there and so on 'til it reaches the root of it all - Object. If not found, then error message appears.
    - Notice the use of the "double underscore" or dunder. The dunder is a reserved word in python - it indicate a protected variable (e.g., __x) or accessing a "magic method" (we come to them later this term), e.g., __lt__.
- **Error-correction on input**: try to keep the end-user from sending empty data or data of the wrong type. Hence when we allow input and we want only a string, cast the input into a string type, e.g.,
    - login = str(input("Enter your login name: "))
- **Concatenation**: with python is funny. Unlike other languages, it's okay to use the + sign because python views all alphanumerics by their number value. E.g., an "A" is 67 (decimal) or 1010 in binary. Note: Codes between 0 - 127 are 7-byte ASCII; 0 - 255 is 8-byte ASCII encoding; there are thousands of encoding methods (win-1265 was the most common for WIndows); today we use UTF-8, a multibyte (1-3) encoding scheme to represent all the characters/glyphs, implementing the standard called Unicode.
- Always use **industrial and professional standards** in your work. For python we'll see about PEP-8 (among others); for language and country codes use ISO639-2. If you're typing on, say, a Chinese version of WIndows, even the letter "a" is going to be stored as 000000000101 (or something like that). If you save a file as "ascii", the leftside set of 8 bits that are all 0s can be discarded without loss of data, making a smaller file.
    - Note: in real practice we need to ensure that all the data we're going to ingest is compatible - so open your files using the optional encoding='**utf-8**'. [Some file reader code automatically may make this conversion but to be sure.]. Also we need to check when reading lines of a file for either the \n or the **\r\n** combination. The old \r was used in Windows (for "return"); the \n for "new line". In all cases if that \r is there the data will screw up.

- **Scope and Visibility**: All languages prefer to have variables created and destroyed as close to where they're used; otherwise they have "global" scope and can be used and altered anywhere in the code.
- **RAM** see more about stack and heap. There's a limited amount of RAM. The low end of ram holds OS commands, global variables, declared but uninstantiated variables, and statics (variables that won't change while the program runs) - there's a limited about of this area ("heap"). The values and ways to fetch the data live in the expandable "stack". So if you declare "x = 5", the "x" is in the **heap**; the 5 is in the stack. Function declarations reside in the heap; that's why lambda functions are better 'cause they act like variables in the **stack** and when they loose focus they're automatically destroyed, freeing up the RAM space.
- **Control of Flow**:
    - Get the logic of your idea down first - then focus on the code. When you're ready to code, it is often better imho to be verbose - write it out and perhaps use hard-coded values instead of variables. That way you can first make sure the logic of the script is correct, and then variable, to make sure the data are correct. Trying to fix the logic and fix the data at the same time can be a mess. Finally, when the logic is correct then look for a python shortcut - python is FULL of them. If you can't get the logic down, you'll have a mess with lambda functions and map(). [later this term]
    - **if ... else**
        - nested if statements. Previously python lacked a "case" structure - the latest version has one, called match. Rule of thumb: if you have 5 or more if statements, rethink the logic of the code or use a different structure (like match).
    - Python's else is weird - can be used after a try-catch block. If you've coded in Java or C++ you'll see this as weird, but it's useful.
    - **for** and **while**: For loops are useful 'cause we know the start and stop; while loops are useful when we don't know the end and need to test for the end condition.
    - **break/continue**: when we wanna make sure we jump out of a loop, use break; sometimes you'll encounter true situation (you've found what you want) but want to keep looping - use continue.
    - As we'll see you want to avoid nesting while loops. [Big O and Big Theta issues later this term]
    - Sometimes starting at the end and working to the beginning is useful. Counting backwards - like a palindrome but also in some records the "file reader" or some other pointer is already at the bottom of the data set - so count backwards each record 'til we have no more to do. You'll often see "while true…" kind of situations.
- **Modularity**: start to practice this now! Learn to break down a task, even what looks really elementary, into understanding the steps involved. That'll help you know the logic, the choice of control-of-flow statements, anticipate the need for placeholder variables and counters, errors. Modularization is the key to good software design; the heart of systems analysis; the key to object-oriented programming.

End of the review points.
file name: Week-2-mainPoints.rtf as of 9/3/22, 8:49 AM