CS 1110 - Summer 2018

- intro to programming -- how to think like a robot :)
- we use the Python* language (*www.python.org*)
- programming environments (many choices):
  - ***Pycharm*** (free from *www.pycharm.com,* which is what I'll use in class), or ***Eclipse*** (free from *www.eclipse.org*), or ***Idle*** (comes free when you download Python), or use your favourite text editor together with the terminal window (ok to start with, but messy later), or ***Komodo Edit*** (the 'edit' version *www.activestate.com/komodo-edit* is free and sufficient -- the full version is expensive and not needed)
- text resources (free):
  - *Introduction to Python 3 on the pycharm website (be careful to use this, not 2!!!)*
  - *They have many tutorials, and give hints on the right for what to enter in the highlighted code boxes (you'll get weird error messages if you run the code as is!)*
  - *docs.python.org/3/*
- LOTS of programming and problem-solving :)

* we'll use 2018.1 for this course, though I will mention the differences from the old 2.7.x for those interested

- **Variables and Types** ... (look up *docs.python.org/3/reference* for precise details)
  - Variables are simply names (or pointers) to values, and these *assignments* are made via

```
first_var  =  42
second_var  =  'this is a string'
first_var  =  'this is another string'
```

Python is dynamically *typed, so the same variable name can point to all sorts of values -- this can be dangerous, so it's up to you to be careful!!*

these = signs don't mean equals, they tell the computer to assign the LHS as a label for the value of the RHS. To test for equality we'll use == but more on that later.

  - Some of the main types (of values) are
    - Numerical, eg integers and floats. There's even a formulation for complex numbers, so that the complex number z = 3 + 4i would be created by `z = 3 + 4j` .
    - Logical, ie boolean: `True` , `False` (technically this is a kind of integer in Python).
    - Sequences, e.g., strings, tuples, and lists. Strings are surrounded by pairs of single, or double, or triple quotes. Note that special characters like ' or " can be created via `\'` and `\"` , e.g., `question = 'How\'s that?'`. You can also force a newline by `\n` and a tab by `\t` . Tuples are created by a comma-separated sequence of things surrounded by ( ), and lists by a comma-separated sequence of things surrounded by [ ]. (Strings and tuples are *immutable*, so the values can't be changed after creation*. Lists are *mutable*.)

* *What this means is that what's been collected can't be changed, although the actual* values *of any of those things can be changed.*

- **Operators** ... (look up *docs.python.org/3/reference* for precise details)
  - The principal arithmetic operators are `+` , `-` , `*` , `/` , `//` , `%` , and `**` . Addition, subtraction and multiplication are obvious, but there are some oddities with the others. *(There are hierarchies of orders of operation, but it's vastly safer to use brackets to force the issue!)*

    `9 / 4` produces `2` in Python 2.x and `2.25` in Python 3.x (to get the answer `2` in Python 3.x requires `9 // 4`). `9.0 / 4` will give `2.25` in both versions. The answer `2` occurs because the computation was being done in the 'type' of integers.

    `9 % 4` produces `1` since it calculates the remainder after dividing by `4` .

    `9 ** 4` produces `6561` since it calculates `9` to the power `4` .

  - Acting on strings we can use `+` between two strings, and `*` between a string and an integer.

    `"it's" + "raining"` produces `it'sraining` and `"sun " * 3` produces `sun sun sun` .

- **Comments** ...
  - "*Code without useful comments will get zero credit!" Even if the code is brilliant !!!!!!!!!*
  - Comments are designed to help people (including yourself next week, next month, next year) understand your code.
  - The relevant symbol is `#` and applies to everything after it on that line.

    `answer = 83 // 10 # note that this is integer division` -- the stuff after the # sign is ignored by the computer

- A simple example program:

```
print("Anyone alive?")  # easy print statement to check that the computer is alive!

print("Please enter an integer ...")     # printing a request to the screen
response_1 = input( )                    # gathering the string entered at the keyboard
print("Please enter another integer ...")
response_2 = input( )                    # note that input( ) works for 3.x, but for 2.7.x use raw_input( )

arith = int(response_1) * int(response_2) # converting the strings to ints and multiplying them

print("The product of " + response_1 + " and " + response_2 + " is " + str(arith) ) # converting the int arith to a string
```

  - this could be saved to a file such as *multiplier.py* and then run in the terminal window or via your favourite IDE (aka integrated development environment).

- Some recommendations for how to write programs:

  1. sketch out on paper what you want to do, don't simply start at the computer
  2. plan out the larger program, but then break it down into simpler parts, building each of them and testing each of them independently
  3. use print statements to see if the values of your variables are what you expected
  4. don't try to build the amazing-ultra-super program as your first version!!! Think in terms of successive versions; the first being pretty basic, and later ones indulging in successive refinements, turning in the most recent version which works!
  5. SAVE all your work, not just on your computer (they die at the most awkward times!!!).

Control ...

- Often we'll want to make a program do different things depending on some condition (for example: go outside if it's nice, stay indoors otherwise).

```
if temperature > 25 :
    print("I'm going outside")
    print("Will you join me for an ice cream?")
else :
    print("I'll wait indoors until it's nicer")
```

*The colons are important here. Also note that unlike many other languages, the test condition isn't in parentheses, and there's no bracketing of the code to be performed -- that part is done by indenting. It's important to note that Python distinguishes between tabs and spaces, so it's best to stick to using 4 actual **spaces** for each level of indentation.*

It's not necessary to have an 'else' block if you don't want one. Also, by successive indentation it's possible to have nested if-else situations, although use of the elif command helps avoid a surfeit of indentations! Multiple conditions can be handled by

```
if temperature > 25 :
    print("I'm going outside")
elif temperature < –5 :
    print("I'll get my skis")
else :                              # note that this will be active if the temp >= –5 and <= 25
    print("I'll wait indoors until it's nicer")
```

The first condition/block which evaluates to true will be the one to become active, any other remaining condition/blocks will be skipped (even if some of them might also be true).

Combining conditions ...

- The comparison operators are `<` , `<=` , `>` , `>=` , `==` , `!=` *(ie not equal to)*
- To combine various boolean-valued expressions, use `and` , `or` , `not` .
- Note that `A and B` will evaluate to false if `A` is false (whence B won't even be checked). Similarly `A or B` will evaluate to true if `A` is true (without checking B).

- **Repetition, aka iteration** ...
  - Often we'll want to do something a set number of times, or even continue doing something until some condition changes.

```
total = 1
base_value = 3          # the number we're going to raise to some power
power = 100             # the power to which we're going to raise the base_value
number_of_times = 0     # this will be a counter to count how many times we've done the multiplication
while   number_of_times < power :
    total = total * base_value
    number_of_times = number_of_times + 1       # incrementing number_of_times by one
print("The value of " + str(base_value) + " to the power of " + str(power) + " is " + str(total) )
```

  - Instead of the *while* loop above, we can use a *for* loop coupled with the *range* function

```
total = 1
base_value = 3          # the number we're going to raise to some power
power = 100             # the power to which we're going to raise the base_value
for   number_of_times   in   range(power) :
    total = total * base_value
print("The value of " + str(base_value) + " to the power of " + str(power) + " is " + str(total) )
```

    - Essentially, range(100) produces a sequence of integers from 0 to 99 which the variable number_of_times runs through successively, saving having to initialise it and increment it as we did for the while loop. It's worth noting that there are a couple of helpful variations: range(27, 50) gives a sequence from 27 to 49, and range(27, 50, 5) gives the sequence 27, 32, 37, 42, 47, ie in steps of 5, up to and excluding 50.

- We'll say much more about iteration a little later, after we've introduced lists and dictionaries.

- We'll explore now some of the properties of the three main sequence types: strings, tuples, and lists. Properties *per se* comprise *attributes* and *functions* (sometimes called *methods*), and *belong* to these objects.

  - One can think of **sequences** as ordered collections of things (ie first to last) which can be indexed (aka numbered). In Python, the numbering starts at 0 so that a sequence of 20 things will be numbered from 0 to 19.

  - Using the string wordy = "This is a string" as our example, the following are functions and operators common to all sequences.

  - "s" in wordy evaluates to *True* since the string wordy does contain at least one s. (Note that strings, but not tuples or lists, allow checking for subsequences, eg "st" in wordy .) There is also the test "his" not in wordy , which in this case evaluates to *False*.

  - wordy + wordy will produce the string "This is a stringThis is a string" and is called concatenation. Although often very convenient, there's a specific danger if this is overused on immutable things like strings (and tuples), since what actually happens is that the space in memory which holds "This is a string" isn't actually overwritten by the longer string (that would mean *changing* the immutable string), instead a completely new space in memory is created to hold the longer string. As you can imagine, if you wanted to concatenate a lot of long strings, then each successive concatenation would create its own new longer string! In this situation, for strings, it would be far better to create a sequence of the strings to be added, and the use wordy.join( ) since this function first measures how long the final string should be, then allocates memory *once* for the final result, and then copies each of the strings in the sequence into the newly allocated string. *(If intrigued, look up the C source-code for Python -- it's available on the python.org site.)*

- More actions common to all sequences ...

  - wordy * 2 produces "This is a stringThis is a string" as does 2 * wordy . There's one thing to watch out for when doing this for tuples and lists, namely that the copying is *shallow* (more on this later).

  - len(wordy) gives the length of the sequence.

  - wordy[8] has the value "a" since it yields the element of the sequence (string in this case) at position 8, where the first position is indexed at position 0. wordy[–5] has the value "t" since it counts backwards from the end, where the last position is labeled -1. This can be generalised to give *slices*, eg wordy[2 : 9] yields "is is a" ie the elements in positions 2 up to (*but excluding*) 9, wordy[2 :] goes from the element at position 2 to the end of the sequence, and wordy[: 2] goes from the beginning of the sequence up to (*but excluding*) position 2. Finally, wordy[2 : 9 : 2] yields "i sa" , since it starts at position 2 and moves in steps of size 2 up to and excluding position 9.

  - min(wordy) and max(wordy) yield " " and "t" respectively (in the case of strings, using lexicographic ordering to make the decision).

  - wordy.count("s") yields 3, ie the number of times "s" appears in the sequence.

  - wordy.index("i") yields 2, ie the position of the first occurrence of "i" in the sequence. If it's not in the string, then index throws an error (not a graceful -1).

- And now some actions specific to **strings** ...
  - `wordy.capitalize( )` yields a copy of `wordy` with the first character Capitalized.
  - `wordy.endswith( "t" )` returns a boolean value, and `wordy.endswith( "ing" )` does likewise. There's a `startswith` function as well, which behaves similarly.
  - `wordy.find( "is" )` yields the index where that substring starts (or -1 if it's not there).
  - `wordy.isalpha( )` yields True if all the characters are alphabetic
  - `wordy.isdigit( )` yields True if all the characters are digits
  - `wordy.islower( )` yields True if all the characters are lowercase, `wordy.isupper( )` is similar.
  - `wordy.lower( )` yields a lowercase copy of the string, `wordy.upper( )` is similar.
  - `wordy.replace( "is", "Gosh" )` returns a copy of `wordy` with every `"is"` replaced by `"Gosh"`.
  - `wordy.split( )` returns a list of the substrings in `wordy` broken by spaces. `wordy.split( "is" )` behaves similarly, except breaking by the string `"is"`. Note that such a list might contain empty strings if, for example, breaking the string `"little"` by the string `"t"`.
  - *For more information and other string functions, read section 4.7 on built-in types in the python standard library at docs.python.org .*

- The second kind of immutable sequence is a **tuple**; typically it's used to hold a collection of things in some order, each of which *can* be different types. *It's immutable in the sense that once built, it can't change what it contains, however that doesn't stop the things it contains from changing* ***their own*** *content.*

  - stuff = ( ) creates an empty tuple, stuff = (a,) creates a tuple with one thing (*note the need for a comma*), and stuff = (a, b, c, d) creates a tuple with four things.

  - The methods listed on pages 7 and 8 as being common to all sequences do apply to tuples, including things like stuff + stuff and stuff * 4 .

  - tuples can contain all sorts of things, even other tuples or lists.*

  - Writing stuff = (a, b, c, d) is called *packing*, whereupon w, x, y, z = stuff is called *unpacking* ; note that there have to be the same number of variables on the LHS as stuff has content.

  - If other is another tuple, then the comparison stuff < other behaves similarly to how string comparison works; the first elements of both tuples are compared, and if equal then the second elements are compared, et segue. If, for example, the first non-equal element of stuff is 'smaller than' the corresponding element of other then the comparison returns *True*, and if, for example, things were equal but stuff turned out to be shorter than other then the comparison again returns *True*.

* *If the (outer) tuple contains things whose content can change (eg a list called goofy), then even though that tuple must still contain goofy (since it's immutable), the* content *that goofy points to can change, eg goofy = [2, 4, 6] and stuff = (1, 3, goofy), then stuff is (1, 3, [2,4,6]) , and if goofy becomes [2, 99], then stuff becomes (1, 3, [2, 99]) .*

- **Lists** are the third kind of sequence we'll deal with; typically they're used to hold a collection of similar things in some order, though they can certainly hold different types. They're *mutable*, so the values being held can be changed, as can the number of things within any given list.

  - `stuff = [ ]` creates an empty list, `stuff = [a]` creates a list with one element (*note that here we don't need the extra comma, although including it still works*), and `stuff = [a, b, c, d]` creates a list with four things.

  - The methods listed on pages 7 and 8 as being common to all sequences do apply to lists, including things like `stuff + stuff` and `stuff * 4` .

  - lists can contain all sorts of things, even other lists. Mutablity even allows fun with slices, eg if `stuff = [a, b, c, d, e]` then doing `stuff[2:4] = [w, x, y, z]` changes `stuff` to `[a, b, w, x, y, z, e]` .

  - We can use *range* to build lists, eg `stuff = [x**3 for x in range(10)]` will yield a list of cubes from 0 to 729. More complicated expressions like the following can be used to build interesting lists: `stuff = [(x, y) for x in (1, 2, 3) for y in (2, 1, 4) if x != y]` . *cf section 5.1 in the python tutorial at docs.python.org* .

  - We could build a 3x3 matrix by `stuff = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` and indulge our love of linear algebra!

  - `del stuff[2]` and `del stuff[2:4]` will delete the 2nd (or the 2nd and 3rd) elements of `stuff` respectively.

- There are also some pretty useful methods belonging to lists ... so if stuff is a list ...
  - stuff.append(x) will append x to the end of the list.
  - stuff.extend(L) will append a whole list L to the end of the list.
  - stuff.clear( ) will empty the list.
  - stuff.reverse( ) will reverse the order of the elements of the list.
  - stuff.remove(x) will remove the first occurrence of x from the list, but it throws an error if it's not there to remove.
  - stuff.pop( ) will remove the last item in the list, and stuff.pop(n) will remove the element in position n from the list.
  - stuff.sort( ) will sort the list (*though we'll learn our own techniques for sorting in this course*).
  - stuff.insert(n, x) will insert x immediately *before* the element at position n in the list. Hence stuff.insert(0, x) and stuff.insert(len(stuff), x) will insert x at the front (or at the back, respectively) of the list.
  - As discussed in class, we can use lists as convenient ways to build other helpful ways of handling data (aka *data structures*), such as *stacks* and *queues*.

- It's time now to see how to delegate tasks to *functions**, how to read and write to files, and how to modularise** things so that we can reuse code, and so code more efficiently.

  - Defining a function is actually pretty straightforward (*cf section 4.7 of the python tutorial*) ...

```
def print_squares(a, b) :
    ''' Print the squares of integers from a up to b '''
    for n in range(a, b+1) :
        print(str(n*n) + ", ")
```

Note that any variables created inside this function, eg a, b and n, have no meaning outside its scope; so eg, if there's an a=9 outside, then that a is unaffected by this function.

This would be called by ...

```
print_squares(2, 9)
```

  - The use of the triple quotes at the top of the function definition actually has great value; it allows the use of `print(print_squares.__doc__)` to print the (hopefully descriptive) text about the function found between those triple quotes (which could span several lines).

  - More involved functions can be built -- indeed, it's often very useful to be able to *delegate* to a suite of functions various actions which need to be performed often ...

```
def get_int_from_keyboard( ) :
    print("Please enter an integer")
    str_input = input( )  #  or use raw_input( ) if python 2.7.x
    return int(str_input)

def which_is_bigger( ) :
    a = get_int_from_keyboard( )
    b = get_int_from_keyboard( )
    if a > b : return (str(a) + " is bigger than " + str(b))
    if a < b : return (str(b) + " is bigger than " + str(a))
    if a == b : return "they're the same size"
```

These would be called by ...

```
wow = which_is_bigger( )
```

which in turn makes 2 calls to

```
get_int_from_keyboard( )
```

each of which returns an int value

Notice the ability to use multiple return statements.

* also called *methods*

** via *modules* and *classes*

- It's also possible to provide default values for function input, so that changing the first line of the print_squares function

```
def print_squares(a=1, b=10) :
    ''' Print the squares of integers from a up to b '''
    for n in range(a, b+1) :
        print(str(n*n) + ", ")
```

allows us to call it as before via print_squares(3, 83), or via print_squares(3) *which produces squares from 3 to the default b value of 10*, or even print_squares( ) *which uses both default values.*

- The special value **None** is returned by a function if there's no explicit value being returned.
- There's great benefit in being able to read from and write to files, and this is made possible via some built-in functions ...

```
my_file = open('amazing.txt', 'r')
```

This will open the file (assuming you have suitable permissions) with *read* privileges. If you want to be able to *write* to the file, then replace 'r' by 'w', note that this will overwrite any previous content. To both *read and write*, use 'r+', and to *append* to the file, use 'a'.

- When you've finished with the file, you should call my_file.close( ) for obvious reasons! The following is actually a pretty slick trick, since it automatically closes the file when done ...

```
with open('amazing.txt', 'r') as my_file :
    # do here all the marvelous stuff
    # you want to do with the file
```

- Once a file has been opened appropriately, then

```
my_file.readline( )
```

will read a line from the file (and subsequent calls to `readline( )` will read successive lines). For our purposes, all files will contain strings, so an *apparently* blank line in the midst of content will contain a newline character, and an empty string will be returned when there's nothing more to read. (*Note that `readline( )` will include the closing (OS dependent) newline character(s) ... to read a line ensuring that any newlines are only represented by '\n', use 'rU' when opening the file.*)

- Looping over the entire file can be done by

```
freds_list = [ ]
for line in my_file :
    if 'Fred' in line :  freds_list.append(line)
```

So this will append the whole line, including the \n, to freds_list if the string 'Fred' is found within that line.

and `my_file.readlines( )` or `list(my_file)` will return a list of all lines (*cf section 7 in the python tutorial for more details*), and `my_file.read( )` will read the file into a single string. *Since some files can be enormous, don't get into an unthinking habit of reading the entire file into memory at once!!!!*

- It's worth making a few comments about *modules* before we embark on *classes* and the underlying principles of object-oriented coding. It's important to design your code to be flexible and reusable, not least because it saves writing pretty much the same stuff every time you need it!! Imagine spending time building code to read and write to files, or gathering input from the keyboard, especially when making it nicely idiot-proof. It makes sense then to put that code into a file which you can import simply each time you need it. That's what a *module* is. The file gets saved, eg as `amazing.py` and can be imported into your programs via `import amazing` . You should get into the habit of *modularising* your code, even if only in simple ways for the moment.

- Classes are a big part of how to make code flexible, modular, and re-usable. The core idea is to think of classes as *manufacturers* of objects, where those objects have properties (*data fields*) and abilities (*methods*). As an example ...

  - Suppose we wanted to be able to build people, then we might have the following class ...

```
class Person :
    ''' This class will allow me to make people!
        object attributes: my_name, my_gender,
                           my_age, my_friends '''

    def person_kind( ) :
        if self.my_gender[0] == 'm' : return 'lad'
        elif self.my_gender[0] == 'f' : return 'lass'
        else : return 'martian'
    def introduce_myself(self) :
        print("Hi, I'm " + self.my_name + ", and I'm a " + str(self.my_age) + " year old " + self.person_kind( ))
    def add_friend(nice_person) :
        self.my_friends.append(nice_person)
    def show_friends( ) :
        return self.my_friends

    def __init__(self, name, gender, age) :
        self.my_name = name
        self.my_gender = gender
        self.my_age = age
        self.my_friends = [ ]
```

This could be used via ...*

```
fred = Person('Fred', 'male', 167)
zork = Person('Qzillarghu', 'vaish', 7)
```

each of which creates a new person, and so allows ...

```
fred.introduce_myself( )
zork.add_friend(fred)
```

This allows me to add friends to the empty list which had been created when this person was built.

These data fields are now attributes for each constructed Person object

This constructs each person, initialising each of the data fields.

Unlike Java, modules in python tend to hold several (related) classes to be imported as needed; close in spirit to a Java package.

* *If the Person class had been saved in a module called People, then the correct syntax would be ...*

```
import People
fred = People.Person('Fred', 'male', 167)
zork = People.Person('Qzillarghu', 'vaish', 7)
```