THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 Week 9 Tutorial

Control Flow and Collections

Switches

Our first example of control flow in Python. Switches act on values and perform an action if the value decays to `True`. They use the `if` `elif` and `else` keywords. The `pass` keyword simply does nothing.

```
if value:
  # Stuff
  pass
elif other_value:
  # Other stuff
  pass
else:
  # More things occur
  pass
```

It's important to note in Python that the `:` symbol denotes a new block of control flow. This block **must** be indented. In python indentation is part of the syntax, and inconsistent indentation (mixing tabs and spaces) will result in a syntax error. It is good to standardise an indent to four spaces.

In addition, scope is shared with indented blocks:

```
if value:
  j = 5
print(j)
```

This leads to problems, where if the block is not executed then the variable does not exist and will cause an error!

A second issue in Python is 'Truthiness' and 'Falsiness'. Consider the following:

```
1 == True # Evaluates to True
1 is True # Evaluates to False
2 == True # Evaluates to False
```

Whereas in C we consider anything not zero to be true. Python resolves this by allowing objects to decay into being True or False upon being cast to `bool`.

So our previous if statement should really be read:

```
if bool(expression):
  pass # Sequence of expressions
```

Question 1: Value Decomposition

For each of the following base Python types see if you can find a value which decomposes to True and another that decomposes to False. For example `bool(int(0))` returns False, while `bool(int(1))` returns True

- `int` (other than 0 and 1)
- `float`
- `complex`
- `string`
- `type` (remembering that type is a type in Python)
- `None`

Functions

As you write increasingly complex and lengthy programs, you may find yourself duplicating certain sections of code quite often. Occasionally you may belatedly notice that you've made a mistake in the code you have just copied half a dozen times and need to fix it.

This code duplication can be avoided by writing functions. Functions are called with some arguments and have a return value. For example the following function 'square' takes some variable 'a' and returns 'a' multiplied by itself.

```
def square(a):
    return a * a
```

This function can then be called.

```
square(2)
4
```

Any Python code can be contained within a function, including loops, switches, and all the other objects that you have seen and used. As any Python object can be passed as an argument, we can also pass collections and functions. As we'll see later.

Question 2: Take it to the Mean

Write a function that takes three arguments. Your function should print the mean of these three numbers and return the number that is the furthest from the mean. If two numbers are equally distant, then return the higher of the two.

Collections in Python

So far you've only seen problems that involve a small number of variables that need to be stored and saved in fixed amounts of memory. However, what happens when the number of variables reaches into the hundreds, thousands or even millions? Rather than writing each variable out on an individual line (a task as pointless as it is time consuming) we instead build a single object that contains other objects: a collection. As objects in Python are dynamically allocated rather than stack allocated we no longer have to worry about constant sizes of stack frames.

The simplest compound object you can imagine is a 'pair'. That is an object that contains two other objects. For constant memory we can just use pointers to dynamically allocated memory for this.

```
struct pair {
  void* first;
  void* second;
}
```

Given a pair you can imagine that one or either of the elements of a pair is itself a pair. If only one element is another pair, and an element of that is another pair then we can construct a 'linked list' of pairs.

```
struct pair {
  void* element;
  void* next;
}
```

We will assume that the final `next` points to `NULL`. Like an array we can index this list using a loop:

```
void* __getitem__(void* first_pair, int i)
{
  void* curr_pair = first_pair;
  while ((i < 0) && (NULL != curr_pair))
  {
    curr_pair = (void*)(((struct pair*)curr_pair)->next);
  }
  return curr_pair;
}
```

Unlike an I can extend this structure to support inserting new elements at an arbitrary position, deleting elements and increasing and decreasing the size of my linked list. In all cases I just manipulate the `next` pointers. The Python implementation of a list is not really a linked list, but it should give you a good understanding of the underlying idea.

A Python list stores a set of items in a single object, and can access them using an index. The first item in the list has index 0, the second has index 1 and so forth. The syntax for initializing a list is:

```
>>> list_obj = [4, "five", 6]
```

In Python, (using an appropriate indirection as might be afforded by `void*`) lists can store any type of variable in any element. The elements of the list can be accessed or modified using an array-like syntax.

```
>>> list_obj[0]
4
>>> list_obj[1]
"five"
>>> list_obj[1] = 3
>>> list_obj[1]
3
```

One very important list is `sys.argv`, which acts in a very similar fashion to our `char** argv` from C.

As we've seen with strings, the `len` builtin acts on lists to return the number of elements in the list (as opposed to a C array where the length is indeterminable).

While we're here, let's look at a few other methods for our lists.

```
list.append(self, object)
# Adds an object to the end of the list

list.insert(self, index, object)
# Inserts an item

list.remove(self, object)
# Removes the first occurrence of object from the list

list.copy(self)
# Copies the list

list.clear(self)
# Removes all items from the list

list.pop(self, index)
# Removes and returns an item from an index
```

There are more methods, which you can see for yourself using `dir`

List and Collections Builtins

We've previously discussed the `len` builtin for strings, upon seeing it for lists, you can probably guess that it's another dispatch method associated with certain types within Python.

```
len([1,2,3])

[1,2,3].__len__()
```

We have the usual `__repr__` and `__str__` dispatches, along with a few that might be a bit surprising.

```
list.__add__(self, list_obj)
# Concatenates two lists

list.__mul__(self, integer)
# Duplicates elements in a list n times
```

We also see the return of `__setitem__` and `__getitem__` from last week. `__contains__` searches a collection for an element and returns `True` if it exists in there, this is actually the arrival of the `in` keyword.

```
'a' in ['a', 'b', 'cde']
['a', 'b', 'cde'].__contains__('a')
```

Loops

Of course once we have a collection of variables, we need some easy method of accessing them all, for this we have loops. Loops delineate some section of code that is repeated until some condition is met. The simplest of these is the *while* loop:

```
a = 0
while a < 10:
    print(a)
    a = a + 1
```

In the above code, so long as the value of *a* is less than 10, then the code in the indented block will run. When the end of the indented block is reached, the condition is checked again.

In Python (but not other languages) loops can also be called directly on collections. For this we use the *for* loop.

```
a = ['test', 2 'silly walk']
for word in a:
    print(word)
```

Of course, this doesn't make much sense does it. The key here lies in the `__iter__` method. This returns an iterator object, the definition of which we will gloss over here. The `iter` and `next` dispatch methods are associated with the `__iter__` and `__next__` methods.

Question 3: Entscheidungs

Reconstruct the Python `for` loop on a list using the `__iter__` method to create an iterator, and its own `__next__` method.

Notice how there is no `__len__` method for the created iterator, there is however `__length_hint__`. Can you change the size of the object you're iterating over while iterating?

Question 4: Hovercraft

Write a Python program called *hovercraft.py* that reads program arguments. Your program should loop through all arguments and check if any of the arguments are the string 'eel' or 'eels'. If all the arguments are eels then your program should print "My hovercraft is full of eels.". If there is at least one eel but your hovercraft is not full of eels then you should print how many eels are in your hovercraft.

If there are no eels, then you should print how disappointed you are at the lack of your slippery friends.

Question 5: Reading between the Lines

The `open` function will open a file in Python. This file object has a number of methods. Of particular interest is `readlines` and `readline`.

Using loops, read and print each line in a file using the readline() function. Then treat the file as a collection of lines and try to loop over it. What happens when you try to print the elements you are looping over?

Question 6: Highscores

You will need to categorise and sort the highscore for each player. Using enumerate function, you can show the current rank of the player. We want to ensure that the player with the highest score is the first element of the list and the player with the lowest score is the last.

Hint: You should sort elements in the list yourself, don't use the `sort` method. Look up `bubblesort` if you're in need of a fast to implement, but inefficient sorting algorithm.

Given input:

```
Player1 5000
Player2 9000
Goku 12000

1. Goku 12000
2. Player2 9000
3. Player1 5000
```

Tuples

Tuples are a category of collections in Python and are virtually identical to lists, except the elements are immutable. That is to say that once set, they cannot be changed.

Tuples are initialised by ending the value allocated to a variable with a comma. Just as with a list, the elements in a tuple are accessed via an index. For example:

```
a = 1,
print(a)
print(a[0])
```

Try to change the value associated with 'a[0]' what error does this throw?

```
a[0] = 2
```

For neatness, and consistency, tuples are delimited by brackets, which helps to distinguish them from lists.

```
a_tuple = (1, 2, 3, 4)
a_list = [1, 2, 3, 4]
print(a_tuple, a_list)
```

Question 7: Slightly Less Useful Lists?

Write a Python program that reads from a file and constructs a tuple where each element of the tuple is the next line of the file. Then convert this tuple to a list by looping over its elements.

You should be able to do each of these operations in one to two lines. If you haven't already, rewrite your approach to use more 'Pythonic' loops.

Tuples and Functions

We lied a little bit about how to define a tuple.

The curved brackets are a convention and aren't strictly required, commas are the actual basis.

```
a_thing = 1,2,3
```

Given only one object can be returned from that function, and a tuple is an object, we can return an arbitrary number of objects from a function by wrapping them in a tuple.

```
def func():
    a = 2
    b = 3
    return a, b
```

Dictionaries

With lists using braces and tuples using brackets, it follows to reason that there must exist another form of collections that use the curly braces '{'. As it turns out, there is: the dictionary. This is quite clearly an associative array, and is in many ways the spiritual base type of all objects in Python.

Dictionaries are a collection that uses 'key-value' pairs. Rather than having an index set by a number, the index is instead set by a key which is then associated with the value that you are actually trying to store in the collection. If you think of a physical dictionary, you search for a particular word and are given its definition. In Python dictionaries the word is the 'key' and the definition is the 'value' that you are storing.

The size of a dictionary can be grown arbitrarily as new keys are used to add new values to the dictionary. If an element is accessed using the key, the value can be modified or changed.

```
a_dictionary = {}
a['key'] = 'value'
print(a['key'])
a['another_key'] = ('tuple_values!', 'and again!')
print(a)
```

This should all seem very familiar to you from last week. After all `globals()` is also a dictionary! In fact most python objects can be considered as dictionaries, whereby if a property such as `__add__` is `in` the dictionary, then the appropriate dispatch method may be used.

Question 8: Configuration File

You have a file containing a number of configuration options. Comment lines are prefixed with a # symbol. Other lines contain the name of a variable followed by its value. If the value is in quotes it is a string, otherwise it is an integer number.

You are to read the configuration file and create a dictionary where the keys are all the names of the configuration options, and the values are the associated values from the configuration file. Some lines may be completely blank and should be ignored. A sample configuration file is given:

```
# Config file
distance_render 'On'
occular_shading 'Off'

# Garbage values
flying_pigs 3
entangled_qubits 3
```

Question 9: Loops and Dictionaries

Given the following dictionary try looping over and printing out each element of the dictionary.

```
a_dictionary = {'q':'Topological entanglement',
'torus':'t',
'area law': 1,
1:'area law',
2:3,
2.4:'double'}
```

What is being looped over in the dictionary? How does this differ from looping over a tuple or a list?

Question 10: Mute and Mutability

Functions in Python treat collections differently from other arguments. If a collection is modified within a function call then the change persists. Changes to characters, integers and non-collection types do not persist. Given that tuples are already immutable, this does not apply to them.

Write a function that takes a list argument and increments each element by 1.

Write a separate function that takes a non-list (integer) argument and increments it by one.

```
list_obj = [1,2,3,4,5]
int_obj = 6
```

Find a way to ensure that the values of the objects in both cases have changed after calling the function.

Question 11: List Adder

Write a function that takes two list arguments. If both lists are of the same length then the function returns a new list comprised of each other list added elementwise (element by element).

If the lists are different sizes it should print some warning indicating this.

Here are some basic test cases, you might want to consider writing some more of your own.

```
test_case_a = [[1,2,3,4], [1,2,3,4]]
print(list_add(a[0], a[1]))
test_case_b = [[1,2,3], [1,2,3,4]]
print(list_add(a[0], a[1]))
```

Args

Let's consider generalising the notion of an argument to a function using a collection.

```
def sum_list(a_list):
  my_sum = 0
```

```
    for i in a_list:
      my_sum = my_sum + i
    return my_sum

sum_list([1,2,3,4,5])
```

Mixing brackets is somewhat ugly and difficult to read (hence difficult to debug), luckily Python has a notion of 'args' as a general argument that accepts an arbitrary number of inputs and then combines them into a single tuple. that can then be accessed.

```
def sum_args(*args):
    my_sum = 0
    for i in args:
      my_sum = my_sum + i
    return my_sum

sum_args(1,2,3,4,5)
```

An existing list or tuple can be expanded into a set of *args when the function is called using the same * operator.

```
a = [1,2,3,4,5]
a_sum = sum_args(*a)
```

This gives us the flexibility to pass lists or individual arguments to the same function.

Question 12: Take it to the Mean 2

Write a function that has a single *args input and does the following:

- Prints each number that has been entered
- Finds and prints the mean.
- Returns the argument that is the furthest from the mean.

Keyword Arguments and Default Arguments

Sometimes you want certain arguments to have a default value, and can then be overridden if necessary with another value.

```
def powsum(*args, power=2):
    n_sum = 0
    for i in args:
      n_sum = n_sum + (i ** power)
    return n_sum
```

Question 13: Take it to the Mean 3

Modify your code from the previous section to add a keyword argument 'print' that is initially set to false. If print is false then the mean is not printed, but the code returns normally. If print is true then the mean is printed.

Kwargs

Of course, a set of keywords and values, is just a dictionary, isn't it. All functions can then be rendered as:

```
def funct(*args, **kwargs):
  args[0] += 2
  args[1] -= 3
  kwargs[name]
  return
```

And indeed, this forms the basis of all functions in Python. We can mix these with our regular function arguments so long as we preserve the order, arguments, args, keyword arguments, kwargs.

```
def powsum(
  n_sum
  argument_b,
  *args,
  keyword=2,
  power=2,
  **kwargs):
  n_sum = 0
  for i in args:
    n_sum = n_sum + (i ** power)
  return n_sum
```

Question 14: (Extension) Sorter

Write a function that takes a list as a single argument and sorts the elements of the list from smallest to largest. Remember that changes to collections within functions persist outside the function call.

For a simple (but inefficient) approach to sorting, look for the bubble sort algorithm.

Question 15: One line ROT

ROT is an encoding algorithm that asks for a rotation of text. Given that you are trying to encode messages you are sending between you and your friend, you agree on using ROT3 as a way of sending encoded messages to each other and being able to decode them.

Example Encoding:

```
Hey Friend! #String for input
Khb Iulhqg! #Encoded string in ROT3
```

Write a program that takes three command line arguments. The first specifying whether to 'encode' or 'decode', the second the rotation number and the third message in question. You are to write a program that implements the ROTN algorithm. It is recommend that you start off by writing an algorithm for ROT13 and then adapt it to allow for any rotation number. The rotation number should be provided as input.

```
Please provide a rotation number: 3
Hey Friend!
Khb Iulhqg!
```

- Check that you are reading the 'encode' and 'decode' strings properly, your program should print an error message and exit gracefully if neither of these modes are specified.
- Ensure that you have properly converted the rotation number to an integer format.
- Combine all the strings from the third argument onwards into a single string.
- The rotation code itself simply increments a letter by the number used. So if the letter is 'A' and the number is 1, then it will give 'B', if the letter is 'A' and the number is 10 then the output will be 'K'. The **ord** and **chr** functions will be useful here. If you are stuck, try working with just a single character and manipulate it.
- Consider what happens when you rotate past the end of the alphabet. The modulus function and something to capitalise letters will be helpful here.
- How do you handle spaces?
- Convert and print your message, make sure you can encode and decode to the original message correctly.

Question 16: ROT Breaker

Similar to the previous problem, you are given an encoded string and a decoded string this time you're going to need to loop through the rotation numbers until you find the one used with this encoding.