Welcome!

COMP 3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 1 DAY 2: PYTHON FUNDAMENTALS

Agenda

1. Questions/Review
2. Mutability in Python
3. Sequence types
4. Lists
5. Tuples
6. Dictionaries
7. Iteration and views

Questions/Review

__name__

__name__ is a built in variable in every python module (file)

Its value is set automatically

__name__ is set to the name of the current module

__name__ == ‘file_1’

__name__ == ‘file_2’

__name__

But as soon as code begins running from a file, the __name__ of that file evaluates to '__main__'

That file's name remains '__main__' for the duration of the program

__name__

But as soon as code begins running from a file, the __name__ of that file evaluates to '__main__'

That file's name remains '__main__' for the duration of the program

__name__

Python doesn't have a main function ☹ We'll need to 'hack' one in

Since we know __name__ == '__main__' from the file we're running, we can 'hack' our code to have a main function

```
If __name__ == '__main__'
  main() #execute our main function
```

Importance of indentation

- Some of you were wondering why we needed the following in our calculator class

```
// calculator code somewhere above
if __name__ == ‘__main__’
  main() #indented
```

- Since the following code also worked just as well

```
// calculator code somewhere above
main() #no indent and no if
```

Importance of indentation

- Everything on indent level 0 that's a code expression will execute
- Your calculator code worked fine because it was one program in one file
- If you needed to import code from another file, all code on indent 0 in the other file would also execute

indent_example.py, indent_example_2.py

SLACK!

Join Slack now – Let's take 5 minutes to do that

Our Slack workspace is called

https://join.slack.com/t/comp3522-bby-wint2020/signup

Use Slack for all communication. If it is something important, send me an email as well.

Sign up and send me and Eric (Eric Qiu2006) a private message right now!

Hi my name is *Your first name and last name* student number is *A0XXXXXXX* git name is *your git name*

Decimal places, extra curly braces

```
my_string = 'hello {0} {{0}}, it is the year {1}'.format('class', 2020.123)

print('Course %s is in %.2f\n' % (name, room)) #decimal places with .2f

print('Course %s {0} is in %.2f\n' % (name, room))
```

Questions.py

Mutability

im·mu·ta·ble

/i(m)ˈmyo͞o͞dəb(ə)l/

Adjective

From the Latin roots “in” (not) and “mutabilis” (to change)

unchanging over time or unable to be changed

String are Immutable

Writing or **altering** individual characters of a string variable is **not allowed**

Strings **cannot change once created**

Instead, an assignment statement must be used

```
word = ‘supercalifragilisticexpialidocious’
word = word.title( )  # this allocates new memory (and un-references the previous one)
print(word)
```

Python Immutable Types

int( )

float( )

complex( )

str( )

tuple( )

bytes( )

frozenset( )

Everything else is mutable (I think!).

Assignment Is Not Mutation

When we assign a value to a variable, we are actually assigning the reference to that value's location in memory

We will review this in some detail later in the term

For now, we can test this like so:

```
id(2.5)
a = 2.5
type(a)
id(a) # same address as id(2.5)!
a = a + 0.0456
id(a) # Not the same – a contains the address of a new float!
```

Memory.py

Phew! Right, that should be enough of strings. Let’s move on to Sequences!

Sequence Types

PROBABLY THE THING PYTHON IS MOST FAMOUS FOR

Sequence Types

Sequence is the generic term for an ordered set

There are several distinct kinds of sequence types in Python

1. str (the string)
2. **list**
3. range
4. tuple
5. bytes

These are not the same thing.

They are stored differently and are treated differently, but they can be processed similarly

Only <u>some</u> sequence operations are common to all

Common Sequence Operations

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
`s.count(x)`	total number of occurrences of *x* in *s*

* Well almost.

Range only supports item sequences that follow specific patterns, and hence doesn't support sequence concatenation or repetition

Mutable Sequence Operations

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from *s* (same as `del s[:]`)
`s.copy()`	creates a shallow copy of *s* (same as `s[:]`)
`s.extend(t)` or `s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i]` is equal to *x*
`s.reverse()`	reverses the items of *s* in place

Containers

A **container** is a data structure used to group related values together

A kind of compound type

A container contains references to other objects

A **list** is a container created by surrounding a sequence of variables or literals with square brackets [ ]

It's like an array, but since it's an object it has useful methods available. (It's similar to but better than Java's ArrayList)

List

This line of code creates a new list variable my_list that contains the two items, an integer and a string:

```
my_list = [10, 'abc’]
```

A list item is called an ***element***

A list is ordered, and indexed. Just like arrays in other languages.

```
my_empty_list = []
```

A list can have many different types of elements. That is, its**heterogenous**.

```
student_data = [“A022343234”, [99,70,85.0], ‘P’]
```

List

Lists are useful for reducing the number of variables in a program (a single list can store an entire collection of related variables)

Individual list elements can be accessed by their index

A **list's index must be an integer**.

```
>>> student_data = [“A022343234”, [99,70,85.0], ‘P’]
>>> student_data[0] #A022342334
```

Try this:

```
>>> student_data[-1] #P
>>> len(student_data) #3
```

Lists are Mutable

Let's try some of these list methods!

```
list_name.append(value)
list_name.pop(index) # remove and use value
list_name.remove(value) # remove by value
list() (Eg: spelling = list("ComplicatedWord")
print(list_name)
list_name[1] = 'new updated value'
del list_name[3]  # remove by position
```

list_examples.py

```
list_name.sort()    # does this change the original list? Or does it return
                                a new instance?

sorted(list_name) # how does this differ from list.sort?


len(list_name)


reverse(list_name)


dir(list)
```

list_examples.py

Sequence Functions and Methods

Sequence-type functions are built-in functions that operate on sequences like lists and strings

Examples include len( ), sorted( )

Sequence-type methods are methods built into the class definitions of sequences like lists and strings

Examples include sort( ), append( ), clear( ), insert( ), pop( ), remove( ), and reverse( )

What's the difference?

Membership Operator

A common task is to determine if a container contains a specific value

Python has membership operators **in** and **not in**

These operators return true if the left operand matches the value of some element in a container

```
>>> colours = [‘red’, ‘white’, ‘cerulean’]
>>> print(‘cerulean’ in colours)
>>> print(‘magenta’ not in colours)
```

Membership Operators

Can be used with sequence types (string, list so far)

- We can check if a list contains something
- We can also use to determine whether a string is a **substring**, or a matching subset of characters, in a larger string:
- So much easier in Python compared to say C or C++

```
request_str = 'GET index.html HTTP/1.1'
if '/1.1' in request_str:
    print('HTTP protocol 1.1')

if 'HTTPS' not in request_str:
    print('Unsecured connection')
```

Tuples

- Think Lists, but fixed and unchanging. That is, it is **immutable**.
- A tuple is also a sequence type, supporting len( ), indexing, and other sequence type functions

```
Eg: pos_vector = (3, 7.0, -39)
```

- Not commonly used, but great when element position and data is usually fixed and unchanging. Eg: location (longitude, longitude)

Tuples (Example)

```
parliament_hill_coords = (45.4236, 75.7009) #tuples use round parentheses
print('Coordinates:', parliament_hill_coords)
print('Tuple length:', len(parliament_hill_coords))

# Access tuples via index
print('\nLatitude:', parliament_hill_coords[0], 'north')
print('Longitude:', parliament_hill_coords[1], 'west\n')

# Error. Tuples are immutable
parliament_hill_coords[1] = 50
```

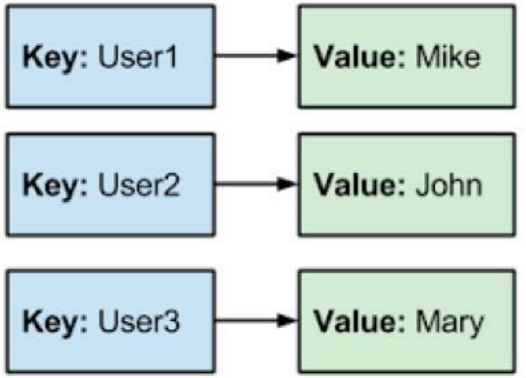
Dictionaries

Similar to other languages, it is a **collection of key-value pairs**

Each key should be unique

```
dictionary_one = { ‘name’ : ‘Lwaxana Troi’ }
dictionary_two = { 1 : ‘COMP3522’,
                   2 : ‘COMP1712’, }
dictionary_three = { ‘Picard’ : ‘SP-937-215’ }
empty_dictionary = { }
```

Key-Value Pairs



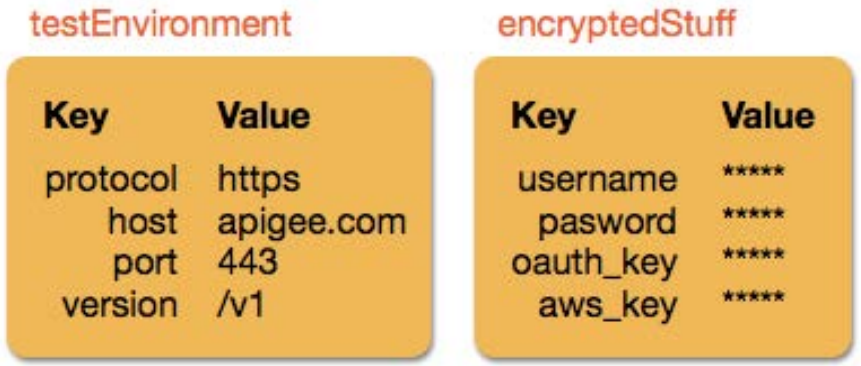
Sometimes keys are called attributes

K-V pairs are everywhere in programming:

- Tables whose contents are arranged and accessed by key
- Query strings in URLS (example.com/lang?name=python)
- Element attributes in markup languages like HTML

Often called:

- Hash table
- Associative array
- (Self-balancing) binary search tree.



What can we use for keys?

We do not want to be able to change the key once it is in the dictionary.

Data is accessed using a key instead of an index

Strings, numbers, tuples (anything immutable)

The keys in a dictionary do not have to be the same type, either!

```
>>> my_dictionary = {}
>>> my_dictionary['value'] = 100.0
>>> my_dictionary[1] = 1
>>> my_dictionary[2] = 2
>>> my_dictionary[2.5] = 'hello'
>>> print(my_dictionary)
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello'}
```

Challenge: what's this?

```
>>> my_dictionary[()] = []
>>> print(my_dictionary)
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello', (): []}
>>> my_dictionary[()] = 'hello world'
>>> print(my_dictionary)
{'value': 100.0, 1: 1, 2: 2, 2.5: 'hello', (): 'hello
world'}
```

Growing a dictionary

The dictionary, like the list, is a dynamic structure (it grows and shrinks as we add and remove key-value pairs):

```
>>> my_character = { 'name' : 'Boaty McBoatface' }
>>> print(my_character)
{'name': 'Boaty McBoatface’}
>>> my_character['occupation'] = 'water bard'
>>> print(my_character)
{'name': 'Boaty McBoatface', 'occupation': 'water bard'}
```

Dictionaries are mutable

We can delete key value pairs too

Use the del statement

```
>>> my_dict = { 1: 'first', 2: 'second', 3: 'third', 3.5: 'third and a halfthth'}
>>> print(my_dict)
{1: 'first', 2: 'second', 3: 'third', 3.5: 'third and a halfthth'}
>>> del my_dict[3.5]
>>> print(my_dict)
{1: 'first', 2: 'second', 3: 'third'}
```

We can use a dictionary in many ways

We can use it to store:

- key value pairs that represent the attributes of some single object, like a character

```
character = {
        ‘age' : 40,
        ‘height' : 100,
        ‘fav_food' : ‘donut’,
        ‘name' : ‘homer’,
}
```

We can use a dictionary in many ways

We can use it to store:

- key value pairs that represent the attributes of some single object, like a character
- a single piece of information about many kinds of objects, like this.

```
exam_results = {
        'abraham' : 50.0,
        'betty' : 60.0,
        'chuck' : 70.0,
        'dolorys' : 80.0,
        'edgar' : 90.0,
        'ferne mae' : 99.9
}
```

Iteration and Views

When a list is not a list but in fact a view

Remember dictionaries?

```
meals = {‘bfast’:’egg’, ‘lunch’:’poké’, ‘dinner’:’spinach’}
keys = meals.keys()
values = meals.values()
entries = meals.items()
```

These methods return **views**, not lists!

```
>>> print(type(keys))
<class 'dict_keys'>
>>> print(type(values))
<class 'dict_values'>
>>> print(type(entries))
<class 'dict_items'>
```

What is a view

A virtual sequence (like the range object we use)

Used for looping

Provides a dynamic 'view' of the entries

- Dynamic: when the dictionary changes, so does the view

Views support three functions:

1. **len**(dictview) returns the number of entries in the dictionary
2. x **in** dictview returns True if x is in the underlying dictionary's keys(), values(), or items()
3. **iter**(dictview) returns an iterator over the view

* https://docs.python.org/3/library/stdtypes.html#dict-views

View example

```
new_dictionary = {1 : 'jeff', 2 : 'eric'}
entries = new_dictionary.items()
print(new_dictionary)
print(entries)

del new_dictionary[1]
print(new_dictionary)
print(entries)
```

View example

```
new_dictionary = {1 : 'jeff', 2 : 'eric'}
entries = new_dictionary.items()
print(new_dictionary) #{1: 'jeff', 2: 'eric'}
print(entries) #dict_items([(1, 'jeff'), (2, 'eric')])

del new_dictionary[1]
print(new_dictionary) #{2: 'eric'}
print(entries) #dict_items([(2, 'eric')])
```

For Loops

```
>>> dishes = {'eggs': 2, 'sausage': 1,
              'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # Use a view for iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504
```

For Loops

Used with sequence types

Will Loop over every element in the given sequence or view

Very efficient

```
for x in sequence:
    do something
```

Like other languages we can **break** and **continue**

For Loops

What do these two pieces of code do?

```
for (name,age) in [(“James”, 22), (“Danica”, 35), (“Rohit”,28)]:
     if age > 30:
           print(“Name: {0}, Age: {1}”.format(name, age))
           break
```

```
for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

For Loops

```
#the classic for loop, iterating over number range
for x in range(0,5):
    print(x)


Output:
0
1
2
3
4 #notice range does not include 5
```

While Loops

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Use while loops when you need to loop over something that is not a sequence.

Remember to update the test expression!

Treat Python With Care

Python Pitfalls

- At first glance python feels more like a scripting language without rigid rules.
- It's "Loosey Goosey"
- It is tempting to use Python to write bad code
- We will come across situations where this flexible nature of python is an asset

That's All For Week 1!

Quizzes start next week!

- The Quiz next week will be during the second lecture (to give you more time to prepare)
- Next week we will cover Ranges and Slicing. This will be followed by Scope. That will give you all enough knowledge to get familiar with Python! (If you aren't already).
- Next week we will also get into OOP concepts and all the good stuff!