Lecture 5: Objects, Modules, and Exceptions

Fall 2020

Topics Introduced: Python object model, modules, and exceptions.

1 Python Object Model

Let's review and elaborate on Python's object model. Key things to always keep in mind:

1. Everything in Python is an **object**.

   An object is a location in memory associated with both a *type and a value*.

2. Variables in Python are **references** to objects.

Simple List Example What happens when we create a list, and then assign another variable to the same data?

```
a = [42, 19, 73]
b = a
```

We would hope that the variable `b` should be "the same" as `a`, in the sense that we expect them to be element-wise equal. But is the underlying data in fact the same? I.e. are we referencing the same object in memory?

```
b[0] = 7    # This assignment modifies first element ref'd by var 'b'.
print(b)    # This mutates data pointed to not only by 'b' ...
print(a)    # ... but also the data pointed to by variable 'a'.
```

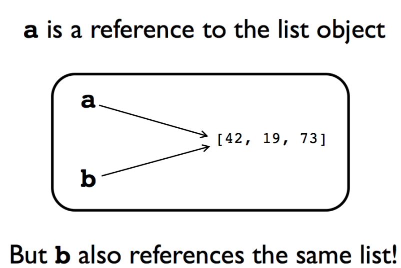

Figure 1: Variables in Python are *references* to objects.

In this example, `a` is a reference to the list object initially set to `[42, 19, 73]`. The variable `b` also references the same list.

References: Analogies This room is like an object.

- "Huang Engineering Center" is an identifier that references where ICME is located.
- "475 Via Ortega Way, Stanford CA" is also an identifier for the same building.

Or perhaps more familiar, we assign identifiers to family members based on their relation to ourselves. E.g. when I say "my sister", I am referring to Monika Santucci.

We emphasize that different identifiers can refer to the same data.

1.1 Python is Dynamically and Strongly Typed

Python is a dynamically typed language, which is in contrast to a statically typed language such as C++. Note that both are still considered *typed* languages.[1]

Statically Typed ⇝ Compiler Safeguards These languages attempt to resolve errors and find potential bugs at *compile time*. If the compiler knows exactly the type of each variable, then the generated machine code can be optimized, resulting in more performant code which potentially executes faster at run time.

Dynamically Typed ⇝ Runtime Errors languages, on the other hand, will delay inspecting the underlying data types that appear in an expression until *run time*. Note that this means that when source code is modified, the interpreter is permitted to dynamically load new code without checking for fool-proof compatibility with all other parts of the program.

Identifiers in Python are dynamic, and aren't necessarily tied to a single type of data.

E.g. an identifier can reference an integer at one point in time, and then later a string.

```python
a = 5
a = 'hi'
```

Strongly Typed ⇝ Implicit Conversions Disallowed Python is also **strongly typed**. The Python interpreter will inspect types at runtime, as they are being executed (since it is dynamically typed), *but* certain classes or errors are precluded from happening (since it is strongly typed): implicit conversions between disparate types is disallowed.

E.g. you can't add a number and a string in base Python, because the language rules state that operation doesn't make sense.[2]

[1] In an untyped language, neither identifiers nor the data which they point to are associated with a type. E.g. the programming language B is untyped: every variable refers to a *word* of memory. Depending on the *context* in which the variables appear, the data will be treated differently, either as an integer value, a string, or a pointer which is to be dereferenced.

[2] This is in contrast to a weakly typed language, such as Perl, wherein the type of one object may be coerced to match another type such that an operation may be carried out. E.g. if you try to use operator `+` on an integer and a string, Perl will coerce the string to an integer (using the additive identity 0) and then perform the addition.

1.2 Assignment: Setting Up a Reference

Everything in Python is an Object This includes numbers, strings, functions, etc.; an object is simply a location in memory associated with a type and a value. The assignment operation, `=`, can be interpreted as setting up a reference. E.g.

```python
a = 'hello'
```

We may pedagogically interpret this as a two part operation.

1. Create a string object containing `'hello'`.

2. Set up the identifier `a` to refer to the newly created string object.

Illustrations and more examples may help.

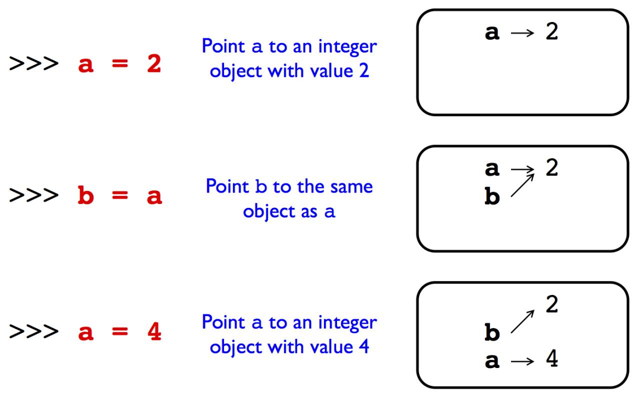

Figure 2: An object is a location in memory with a type and value. The boxed region can be interpreted as a memory layout or diagram. The location in the diagram which a value appears corresponds to the location in memory in which the object is stored. Here, we leave types implicit.

We even implicitly create temporary objects for intermediary values within expressions.

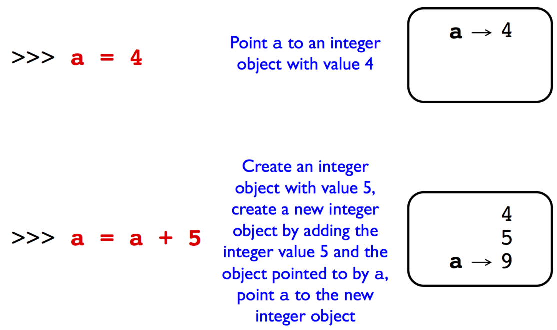

Figure 3: Integers are also objects and must be created in memory. E.g. in order to "add 5" to an object, we first have to define a reference to the integer value 5, then use this data as part of an input to an `add` operation.

1.3 References (Details)

1.3.1 Checking References

We can check if two names (variables) reference the same object with the `is` operator.[3]

```
a = [42, 19, 73]
b = a
print(a is b)  # Returns True.
```

In memory we may visualize the following schematic.

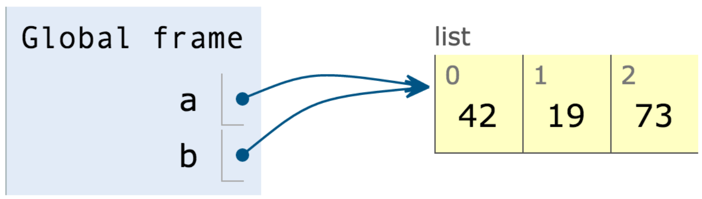

Figure 4: The `is` operator checks to see if two references point to the same object. I.e. it can be seen as a test of *identity*.

```
b = [42, 19, 73]
print(a is b)
```

In memory we have:

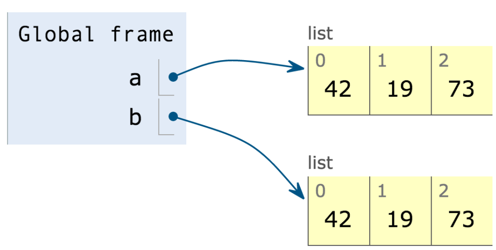

Figure 5: If we assign `b = [42, 19, 73]`, then even though the values are the same, we can really think of the data as being different. Mutating one object will not affect the other.

Check it out in Python Tutor. There's one more nuance to this diagram that we need to refine, and that is how small (magnitude) integers are stored permanently.

1.3.2 Integers and References

Integers are objects also and need to be created in memory. Let's explore this a bit.

[3] Implementation detail in CPython: the `id` operator returns the position in memory where the object is stored.

```python
a = 1024
b = a
a is b      # Returns True: we asked 'b' to refer to same data as 'a'.
```

```python
a = 1024
b = 1024    # Python creates a *new* integer (diff location), also with value 1024.
a is b      # Returns False: 'a' and 'b' refer to different pieces of data.

a = 16
b = 16
a is b      # For small integers, Python reserves persistent storage.
```

In the last code block, `a` and `b` point to the same object, because Python preallocates some integers. This is done as an internal optimization for the Python interpreter written in C.

Preallocated Integers (an Implementation Detail) Python creates permanent storage for integer objects in the range $\{-5, ..., 256\}$, instead of constantly creating and destroying these objects which we anticipate to be frequently used. Integers outside this range are created and destroyed on an as-needed basis.

```python
a = -6    # Here, we create two separate integer objects...
b = -6    # ... in memory. Different locations, but same values.
a is b    # Returns False.

a = -5    # The C-API has written an optimization such that we...
b = -5    # ... don't constantly create/destroy integers in [-5, 256].
a is b    # Returns True accordingly.

a = 256
b = 256
a is b    # Returns True, since [-5, 256] all stored permanently.

a = 257   # Here, we're outside the range of [-5, 256].
b = 257   # Whence this is a different object in memory.
a is b    # Returns False.
```

Of course, this is all very different from the `operator.eq`, i.e. `==`.

1.3.3 String Reuse (Another Implementation Detail)

```python
a = 'hello'
b = 'hello'
a is b      # Returns True. For a longer arbitrary string, may return false.
```

Why are Strings Immutable? Let's consider why we might preference strings to be immutable.[4] An **immutable** object can't be modified. The reasons relate to *performance*.[5]

- Can setup storage for a string exactly once, because it never changes.
- Dictionary keys required to be immutable, so that we can quickly locate keys.

1.4 Containers and Element References

It's important to realize that Collections store other data (structures), i.e. an element of a data structure is simply an object (a location in memory associated with a value and a type).

- Elements in a list, or key and value pairs in a dictionary, contain references to objects.
- Those references can be to "atomic" data types like a Boolean, number, or string. They can also be to more complicated data types, like other containers.
- There are some restrictions, for example we've discussed that the key objects in a dictionary must be immutable (e.g. numbers, strings, or tuples).

```
a = [42, 'hello']
b = a
```

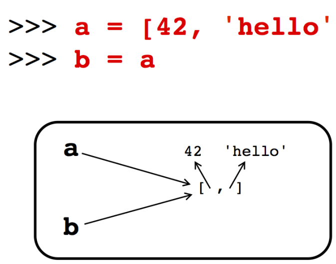

Figure 6: A more accurate memory diagram reveals that individual elements of an Abstract Data Type are simply references to other objects, which themselves can be atomic or another ADT.

1.5 Copying: Shallow vs. Deep Copy

We must emphasize that simple assignment does not give us a copy of the object (e.g. a list), only an additional reference to the same object (list in our example). It's natural to ask: what if we really want an additional copy that can be modified without changing the original?

[4]This is totally separate from "reusing" an object, i.e. wherein two identifiers refer to the same data.
[5]We remark that this is a design decision not uncommon in other languages (e.g. Java strings).

Shallow copy A shallow copy (`copy.copy`) constructs a new list and inserts references to the objects referenced in the original.

```python
a = [42, 'hello']
import copy
b = copy.copy(a)
```

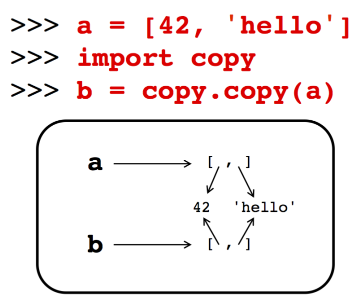

Figure 7: A shallow copy creates a *new list*, where each element refers to the same element referred to by the original list entry.

Shallow Copies and Mutables Note that since a shallow copy simply sets up references to the same underlying objects, then if said objects are mutable we can inadvertently run into results that at first surprise us.

```python
a = [19, {'grade':92}]
b = copy.copy(a)

a[0] = 42             # Modifies first element of 'a' only.
print(b)              # First element of 'b' still refers to integer 19.
a[1]['grade'] = 97    # We modify the dictionary referred to by both 'a' and 'b'!
print(b)              # Accessing 'b["grade"]' returns integer value 97.
```

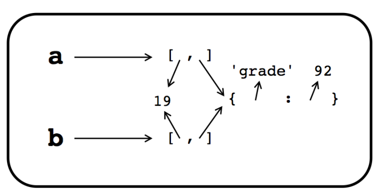

Figure 8: A shallow copy sets up references to the same underlying data of the original container. Since object `a` contains a dictionary, then upon shallow copy to variable `b`, the first element in either variable in fact refers to the same dictionary.

This may not be desireable in all cases, whence there is also a *deep copy* sub-routine.

Deep Copy A deep copy (`copy.deepcopy`) constructs a new list and inserts copies of the objects referenced in the original. It will (recursively) copy all nested data structures.

```
a = [19, {'grade':92}]
b = copy.deepcopy(a)

a[0] = 42              # Request first element of 'a' to refer to integer 42.
a[1]['grade'] = 97   # Request the dictionary ref'ed by 'a' to update value associat
```

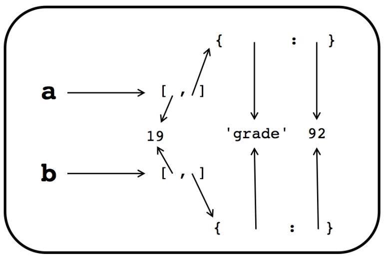

Figure 9: In a *deep copy*, each element is a copy of the object referenced by the corresponding element in the original container. Since strings may be re-used, and since we are only dealing with integers in the set $\{-5, \ldots, 256\}$, datum are not duplicated (i.e. the implementation details don't require there to be additional copies of the string `grade` or the integers 19, 92.

1.6 Tuples and Immutability

Let us revisit the nuances relating to tuples and immutability: the length of the tuple and which objects are referred to by the tuple are fixed from the time of creation.

```
a = [42, 'feed the dog', 'clean house']
import copy
b = copy.copy(a)   # Set up references to same underlying objects.
c = (a,b)          # Create a tuple. Since 42 in {-5, ..., 256} and strings may ...
                   # ... be re-used, then even though the lists are different...
                   # ... the underlying data referrenced is the same.

b[0] = 7           # Here, we ask the first element of 'b' to refer to integer 7.
print(c)           # Of course, first element of 'a' still references integer 42.
c[0][0] = 7        # We can change the first element of 'a' as well...
print(c)           # ... and this will be reflected here.

# c[0] = [73, 'wash dishes', 'do laundry'] --> Error: Tuple object disallows item a
```

The immutable property of tuples only means I can't add or remove elements from the tuple, and I can't ask the elements of the tuples to refer to different objects. However, the underlying objects themselves can be changed if they are mutable.

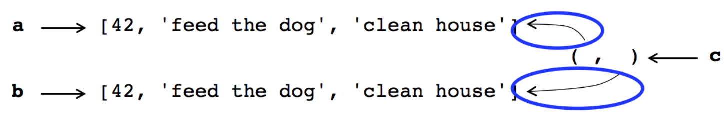

Figure 10: In this memory diagram, the integers and strings need not be duplicated (since the integers appearing are in the set $\{-5, \ldots, 256\}$ and strings may be reused). However, the important part is that while we can never change the fact that `c` is a tuple of length two, where each element is a list, we can modify the underlying lists themselves. E.g. aside from the example pictured, we could also query `[c[0].pop() for i in range(3)]` in order to leave the object referenced by variable `a` an empty list.

1.7 Memory Management

What happens to objects that are no longer referenced?

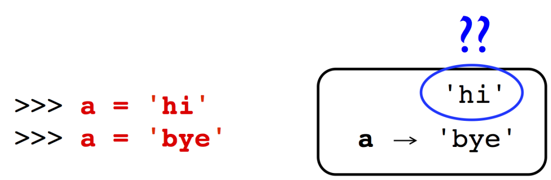

Figure 11: Motivating Garbage Collection: Unlike the integers $\{-5, \ldots, 256\}$, strings aren't guaranteed to persist in a single location in memory, even though both types are immutable. What happens to strings (or any object) when we "don't need them anymore"?

Garbage Collection When an object is no longer reachable or accessible via an identifier, it may be garbage collected; the idea is simple: if an object requires resources but is no longer being used, we may as well free said resources for other parts of our program (or operating system) to use. Python implements garbage collection via reference counting.

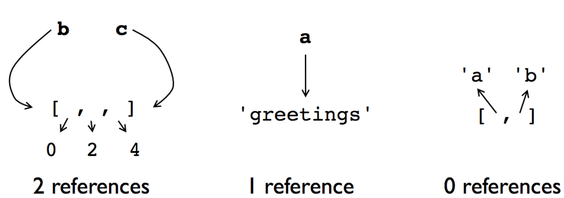

Figure 12: On the left, we depict a case where each integer is referenced by two variables, i.e. lists `b` and `c` (which in this example happen to refer to the same list). In the last example on the right, we have a single unreferenced list, where the elements themselves are "a" and "b". However, the list is not aliased to any variable, whence these character elements are inaccessible. Strings are immutable, but Python doesn't require we store them throughout the program's entirety. Whence we may garbage collect these data (both characters and the list).

1.8 Recommended Reading

Chapter 6: The Dynamic Typing Interlude, from "Learning Python", by Klutz.

2 Python Modules

2.1 Modules as an Organizational Tool

Your code should be organized in some way. We often split code across multiple files to make it easier to maintain and re-use.

For larger projects, we in fact often have multiple directories, each with multiple files.

E.g. consider the grading tools we might use for CME211. We might have a set of sub-routines that lets us fetch student repositories from Github and later push feedback, and it would make sense for this code to set in its own file. On the other hand, we also might have a set of sub-routines that actually perform integration tests on your programs. This grading utility tools have nothing to do with our Git utils, and so we place it in another file as well. But! All the code is useful for this class as a whole, and so we place it within a `CME211` module.

2.2 Using Modules - Practical Details

Code in Python is organized and accessed via *modules*. We've already seen and used examples such as `math` and `time`; they are accessed using an `import` statement.

2.2.1 Basic Usage

Import Here is an example of importing and using a function from the `time` module:

```
import time
time.time()     # Prints out current time in seconds since the Epoch (Jan 1st, 1970)

# time()                  --> Error: time module is not callable.

print(type(time))         # <class 'module'>
print(type(time.time))    # <class 'builtin_function_or_method'>
```

We must keep in mind that the module name/object is different than the function that exists inside of the module.

Reference to a Function Functions are also objects and may be assigned to a variable:

```python
t = time.time
type(t)          # <builtin_function_or_method>
t is time.time   # True.
t()              # Returns time in seconds since Epoch.
```

Everything in Python is an object!

Importing a Single Function We can import a single function from a module:

```python
from time import time
print(type(time))    # <class 'builtin_function_or_method'>
print(time())        # Returns time in seconds since Epoch.
```

Here, we've created an alias for the `time.time()` sub-routine. Another example is `from math import sqrt`.

Renaming We can rename a function in the import statement using `as`.

```python
from time import time as timer
print(type(timer))    # <class 'builtin_function_or_method'>
print(timer())        # Returns time in seconds since Epoch.
```

Wild Card Import We can even import everything from a module into the global namespace with the following.

```python
from time import *     # Use a wildcard character to match against all functions.
print(type(time))      # <class 'builtin_function_or_method'>
print(time())          # Returns time in seconds since Epoch.
```

This is normally not a good idea, because you may unknowingly overwrite some symbols that have been defined elsewhere.

2.2.2 Modules and Namespaces

Modules not only allow us to partition code into separate files; they also provide distinct namespaces. A namespace becomes more important in larger projects, where reuse of common terms can be, at best, confusing, and at worst the root cause of bugs. In general, even attribute renaming and/or wild card imports can make code less readable and more difficult to debug

Example Here we know where `time()` is coming from:

```python
import time
import mymodule
# ...
t = time.time()
```

But, what if we use wildcard imports...does `time()` come from `time` or `mymodule`?

```python
from time import *
from mymodule import *
# ...
t = time()    # Best case: confusing. Worst case: bug.
```

Recommendation: be explicit when using module functions!

2.2.3 Authoring Modules

See file `mymodule1.py`, included below and in our Github.

```python
def summation(a,b):
    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

That's it! There's nothing special about this code. You've seen all the Python syntax before. Simply including code in a file is enough to define a module.

2.2.4 Using Your First Module

From the command line, and working in the `lecture-05` directory:

```
$ python3
>>> import mymodule1
>>> mymodule1.summation(1,100)
5050
>>> mymodule1.primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

2.2.5 Improving Your Module

Add test code in file `mymodule2.py`:

```python
print('Testing function summation():...', end='')
total = summation(1,100)
if (total == 5050):
    print('OK')
else:
    print('FAILED')
```

2.2.6 Testing Your New Module

```python
import mymodule2 as mm2       # Simply import the module invokes our test.
print(mm2.summation(1,100))   # Here, we call the same old sub-routine.
print(mm2.primes)
```

Import Process When you do `import mymodule2` several things happen.

1. The Python interpreter looks for a `.py` file with the same name as the module, starting with your current directory followed by looking in system wide locations specified by the system `path` variable.
2. Code is byte compiled from the `.py` file to a `.pyc` file.
3. File is processed from top to bottom.

It is in this last step that our tests were executed.

2.2.7 Locating Modules

Python searches for modules based on directories listed in the `sys.path` list object. The first item in this list is actually an empty string, which is used to denote the current directory.

```python
import sys
print(sys.path)    # Prints a long list, dependent on your OS configuration.
```

Since `sys.path` is just a Python object, we can manipulate it. Let's remove this directory from `sys.path` and try to load a module (that we have not yet loaded, but which does live in the current working directory).

```python
sys.path.remove('')
# import mymodule3  --> No module named 'mymodule3'.
```

If we add the current working directory back to the object, all is well.

```python
sys.path.insert(0,'')
import mymodule3    # OK.
```

2.3 .pyc Files

We mentioned that when a module is loaded by the interpreter, one of the steps is to process the source code in the .py file into a byte-compiled code which is then stored in a .pyc file.

```
$ ls *.py*
mymodule1.py  mymodule1.pyc  mymodule2.py  mymodule2.pyc
```

Why Byte-Compile? It turns out that .pyc files can be faster to load than a corresponding .py file. Of course, the runtime performance once loaded is identical since one is a direct translation of the other.

2.3.1 __name__ and __main__

There are several *private variables* which are defined each time a Python program is executed.

- Special variable __name__ is equal to __main__ if the file is being executed as the main program.
- __name__ will not be equal to __main__ if the file is being imported.

Let's look at an example to see why this might be useful.

2.3.2 "Hiding" Code During Import

See mymodule3.py.

```python
if __name__ == '__main__':
    print('Testing function summation():...', end='')
    total = summation(1,100)
    if (total == 5050):
        print('OK')
    else:
        print('FAILED')
```

If we were to launch a new Python interpreter and import this module, then the testing subroutine would *not* be executed. If, however, we launch a Python program from command line where the mymodule3 is supplied is main argument, then these tests *will* be executed.

If we want to import the contents of the module without executing the main sub-routine, i.e. the tests in this case.

```
import mymodule3     # Not being run as main. So, tests are not executed.
```

If we wanted to run the test code, we'd do by invoking the module as a main program. From the command line.

```
$ python3 mymodule3.py
Testing function summation()... OK
```

Being able to decompose our programs into main routines which are separate from sub-routines and constants enable us to use a module in different ways. If we want access to the core functionality, we can run the `main` program. If we simply want to benefit from one of the helper functions or constants defined in the module, we can simply import it.

2.3.3 Documenting the Module

This part is essential. Any reasonable code will have excellent documentation. See `mymodule4.py`.

```
"""
My Module: a collection of miscellaneous code.

This module defines a summation sub-routine alongside a list of primes.
These utilities may be of interest to someone with interests lying at the
intersection of number theory and computation.
"""

def summation(a,b):
    """
    Returns the sum of integers  between arguments a and b, inclusive.

    We anticipate the input arguments to be each integers. The output
    under this assumption is also an integer. We remark that there
    are simple closed form solutions for this problem, using the fact
    that summation(0,n) = n*(n+1)/2.
    """

    total = 0
    for n in range(a,b+1):
        total += n
    return total

primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

if __name__ == '__main__':
    print('Testing function summation():... ', end='')
    total = summation(1,100)
    if (total == 5050):
        print('OK')
    else:
        print('FAILED')
```

If you want to access the wonderful documentation that you've written, you can do so via `help`.

```python
import mymodule4
help(mymodule4)
```

```
Help on module mymodule4:

NAME
    mymodule4 - My module of misc code.

FUNCTIONS
    summation(a, b)
        Returns the sum of numbers between, and including, a and b.

DATA
    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

FILE
    /Users/asantucci/git/cme211-notes/lecture-05/mymodule4.py
```

2.3.4 Recommended Reading

Chapters 22 and 23 in Learning Python; Modules: The Big Picture, and Module Coding Basics

3 Python Error Handling

3.1 Syntax Errors

A syntax error is incorrect based on the language rules. Code containing syntax errors cannot be executed by the Python interpreter. Here are a few examples:

```python
# Forgetting to include a colon after starting function declaration.
def my_func()                 # An error will point to the space after my_func().
  print("Hello from my_func")   # ... and will state "SyntaxError: invalid syntax".
```

Or for example if we forget to close a bracket when defining a list. (If we only enter the following into our *interpreter*, it will *hang* in so far as hitting `enter` will yield a new prompt from the interpreter to complete our expression. We must enter a new expression in order to trigger the error.)

```python
a = [1, 2, 3              # SyntaxError: unexpected EOF while parsing
```

Or for example if we forget to include a comma when defining a list.

```
a = [1, 2 3, 4]
#         ^
# SyntaxError: invalid syntax
```

All of the above examples can be spotted by the naked-eye, so long as we know the rules for the language syntax.

3.2 Runtime Errors

Runtime errors occur when syntactically correct code does something wrong (like attempt to access a list out of bounds, or divide an integer by zero). We have seen these before.

```
a = [3, 7]
a[2]          # IndexError: list index out of range

b = {'cupcakes' : 7, 'brownies' : 2}
b['cookies']  # KeyError: 'cookies'
```

These errors don't involve typographical mistakes, but are instead of a different nature. Spotting them requires understanding of what particular data is stored in the object at a particular point in time during the program's execution. E.g. the syntax `a[2]` is of course valid for any sliceable variable `a` which is at least three elements long, and `b['cookies']` is valid syntax for a dicionary `b` containing a key `'cookies'`. How can we recover from such errors? Without error handling, our program will halt!

3.3 Exceptions

Runtime errors generate exceptions, which can potentially be caught. Uncaught exceptions propagate up to the interpreter, which ultimately halts execution and displayes the information in a traceback.

Python uses a try/except model for error handling.

```
f = open('thisfiledoesntexist.txt')   # FileNotFoundError: No such file or director
```

```
try:
    f = open('thisfiledoesntexist.txt')
except IOError:
    print("That filename doesn't exist.")
```

Here, we caught the exception raised when `open` could not find the file. The `try-except` syntax allows us to control what happens when an exception occurs. There is a listing of predefined exceptions in the Python documentation, and additionally you may define your own exceptions.[6]

3.3.1 Catching Multiple Exceptions

Specific exceptions can be handled by specifying the exception type after `except`.

```
try:
    5/0
except IOError:
    print('I/O error')
except ZeroDivisionError:
    print('Zero division error')
except Exception as e:
    # here we get access to the exception object
    print(e)
```

This in fact prints out that we have a "Zero division error".

3.3.2 Raising Exceptions

From `mymodule5.py`:

```
import types

def summation(a,b):
    """
    Returns the sum of integers between a and b inclusive.
    """

    if (type(a) != types.IntType or type(b) != types.IntType):
        raise ValueError('Expected integers as input for summation.')

    total = 0
    for n in range(a,b+1):
        total += n
    return total
```

Using our previos `mymodule4`, invoking our `summation` argument with an integer and a string yields an error that may be difficult to interpret.

```
import mymodule4
mymodule4.summation(1, 'hello')     # TypeError: Can't convert 'int' object to str
```

But with some foresight, we can gracefully handle this exception.

[6] We'll define exceptions via a class, and we'll learn how to define a class at the end of this lecture.

```python
import mymodule5
mymodule5.summation(1,'hello')       # ValueError: Expected integers as input for su
```

3.3.3 Recommended Reading

- Python Tutorial: Errors and Exceptions
- Chapter 33: Exception Basics
- Chapter 34: Exception Coding Details