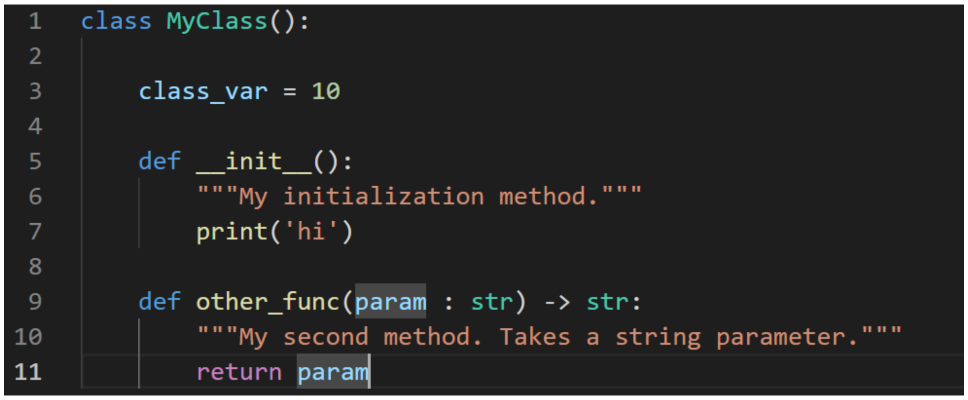
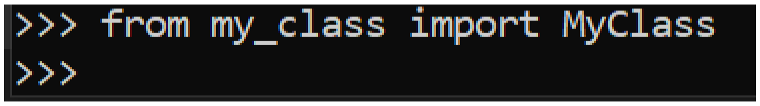
When I first started out in python, I was very intimidated by the interpreter. However, after having programmed in the language for many years, I came to realize that the interpreter (and more specifically the bundled REPL setup) is very useful for exploring unknown codebases. There are many instances where the documentation for a library is not as verbose or robust as I'd like, and diving right in and testing out code directly can save lots of time. But where do you start? This pointer will hopefully help you get a little more comfortable.

First, we'll start by making a class. In this case, the class will solely be used for demonstrative purposes, so it won't do anything fancy.



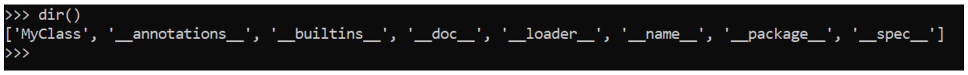
```python
class MyClass():

    class_var = 10

    def __init__():
        """My initialization method."""
        print('hi')

    def other_func(param : str) -> str:
        """My second method. Takes a string parameter."""
        return param
```

Let's go ahead and import this class.



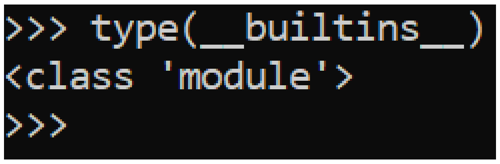
```
>>> from my_class import MyClass
>>>
```

The first REPL command that I'd like to introduce is dir(). The dir() function is a builtin function that allows you to inspect all the names that acre currently in scope (https://docs.python.org/3/library/functions.html#dir). It can be really handy if you're unsure of what the name of something is, or for determining what is loaded into memory. In this case, you will see the default names have been loaded by the interpreter (all the dundered names), as well as the "MyClass" class that we just created.



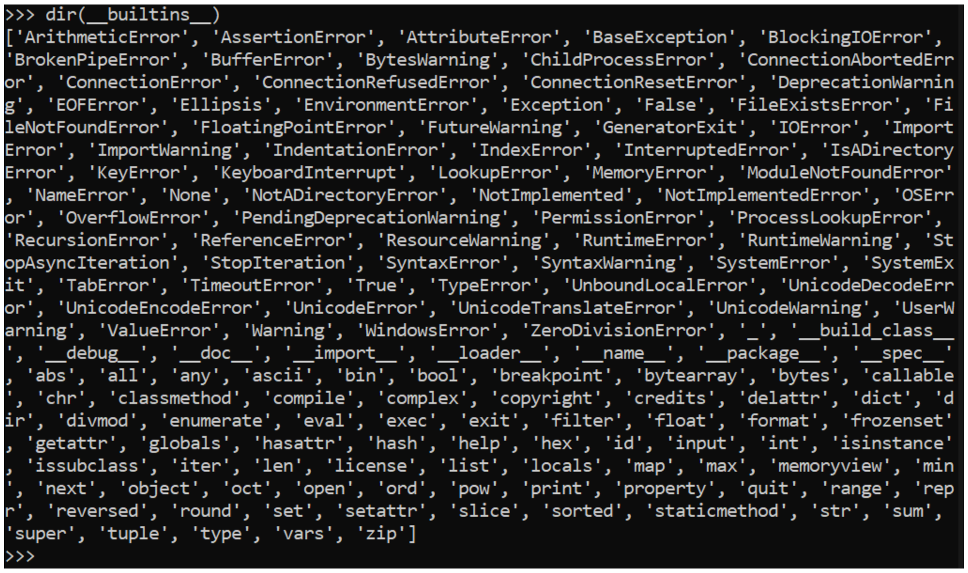
```
>>> dir()
['MyClass', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

We cover a number of these throughout the course (recognize the __name__ variable here?). I won't be covering all of them in detail in this pointer. Let's go ahead and take a look at the __builtins__ name. Since I'm not really sure what that is, let's first determine its type. You've done this before in the course.



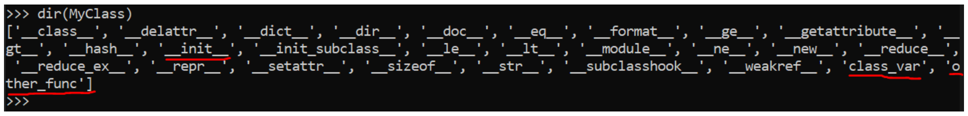
```
>>> type(__builtins__)
<class 'module'>
>>>
```

Well that's great, it's a module. But that doesn't tell us very much. So, let's use the dir() function to interrogate the __builtins__ module to see what names are valid in its scope.



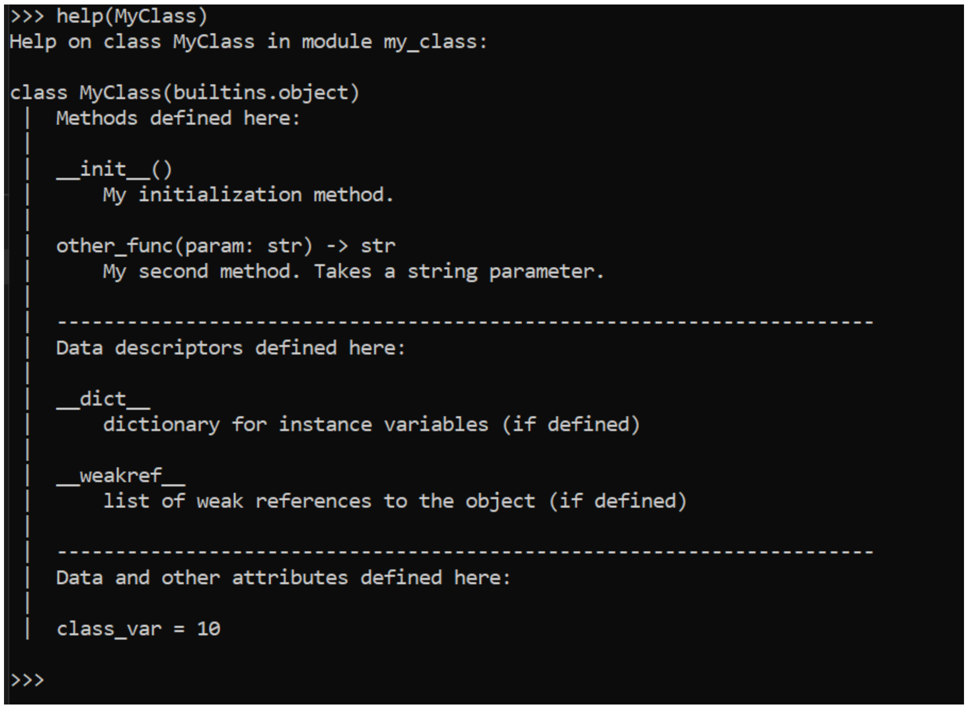
```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedErr
or', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarnin
g', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'Fi
leNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'Import
Error', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectory
Error', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError'
, 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSErr
or', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'St
opAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemEx
it', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeErr
or', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserW
arning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__build_class__
', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__'
, 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable
', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'd
ir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset'
, 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance'
, 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min
', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'rep
r', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
>>>
```

Woah! That's more like it! Python has a lot of built in functions and classes! If you'll notice, you can see all your favorite builtin things like input() and str(), etc. You can even see dir() here! But for now we'll leave that alone. Let's look back at our custom class. You can dir() that right up too!



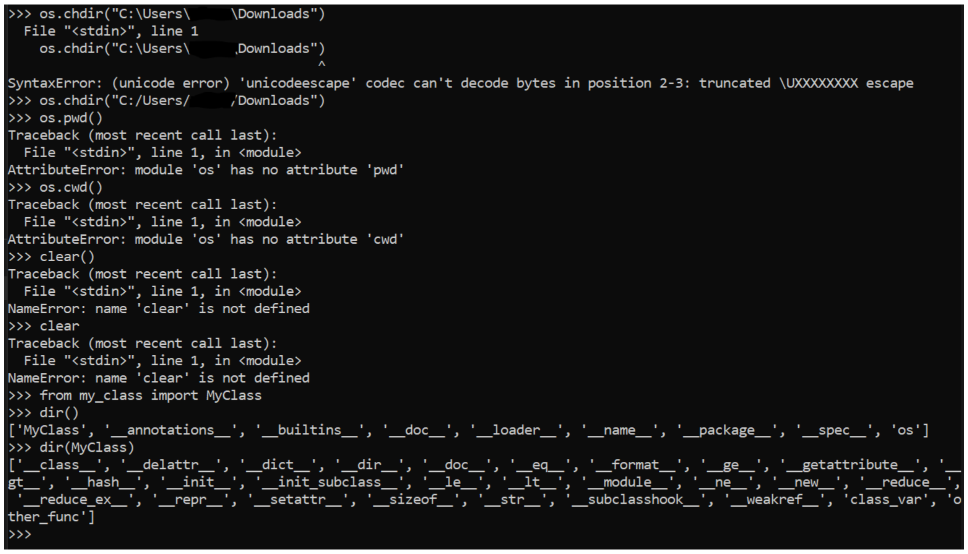
```
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'class_var', 'o
ther_func']
>>>
```

Notice that we see class_var, __init__, and other_func appear in our list of names, as well as more dundered builtin object methods. This is great, but it's not terribly useful, so let's look at another funciton: help. Help will show you the structure of the class, including the methods (including their docstrings and type annotations) and their attributes!



```
>>> help(MyClass)
Help on class MyClass in module my_class:

class MyClass(builtins.object)
 |  Methods defined here:
 |
 |  __init__()
 |      My initialization method.
 |
 |  other_func(param: str) -> str
 |      My second method. Takes a string parameter.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  class_var = 10

>>>
```

“This is great and all, but what if I mess up?” You may ask... No problem! Here is a screenshot of my interpreter as a work in progress for making this example. There is no harm in making a mistake, you can easily keep going. Python will report out any exceptions. (For those wondering, the command I was looking for to list my current directory was “os.getcwd()”.



```
>>> os.chdir("C:\Users\[redacted]\Downloads")
  File "<stdin>", line 1
    os.chdir("C:\Users\[redacted]\Downloads")
                                  ^
SyntaxError: (unicode error) 'unicodeescape' codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape
>>> os.chdir("C:/Users/[redacted]/Downloads")
>>> os.pwd()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'os' has no attribute 'pwd'
>>> os.cwd()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'os' has no attribute 'cwd'
>>> clear()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'clear' is not defined
>>> clear
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'clear' is not defined
>>> from my_class import MyClass
>>> dir()
['MyClass', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'os']
>>> dir(MyClass)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__
gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'class_var', 'o
ther_func']
>>>
```

Running code inside the interpreter can be very useful for testing out a thing or two, but it’s not very suitable for running large amounts of code, as it can get tedious to keep typing things out line by line. But hopefully this example had given a little insight on how you might explore unknown codebases.