merge5

June 14, 2023

0.0.1 Docstrings and Annotations

Docstrings When we call **help()** on a class, function, module, etc, Python will typically display some information:

[1]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(…)
    print(value, …, sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

We can define such help using docstrings and annotations.

[2]:
```
def my_func(a, b):
    return a*b
```

[3]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a, b)
```

Pretty bare! So let's add some additional help:

[4]:
```
def my_func(a, b):
    'Returns the product of a and b'
    return a*b
```

[5]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a, b)
    Returns the product of a and b
```

Docstrings can span multiple lines using a multi-line string literal:

[6]:
```
def fact(n):
    '''Calculates n! (factorial function)

    Inputs:
        n: non-negative integer
    Returns:
        the factorial of n
    '''

    if n < 0:
        '''Note that this is not part of the docstring!'''
        return 1
    else:
        return n * fact(n-1)
```

[7]:
```
help(fact)
```

```
Help on function fact in module __main__:

fact(n)
    Calculates n! (factorial function)

    Inputs:
        n: non-negative integer
    Returns:
        the factorial of n
```

Docstrings, when found, are simply attached to the function in the `__doc__` property:

[8]:
```
fact.__doc__
```

[8]:
```
'Calculates n! (factorial function)\n    \n    Inputs:\n        n: non-negative
integer\n    Returns:\n        the factorial of n\n    '
```

And the Python **help()** function simply returns the contents of `__doc__`

Annotations We can also add metadata annotations to a function's parameters and return. These metadata annotations can be any **expression** (string, type, function call, etc)

[9]:
```
def my_func(a:'annotation for a',
            b:'annotation for b')->'annotation for return':

    return a*b
```

[10]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a:'annotation for a', b:'annotation for b') -> 'annotation for return'
```

The annotations can be any expression, not just strings:

[11]:
```
x = 3
y = 5
def my_func(a: str) -> 'a repeated ' + str(max(3, 5)) + ' times':
        return a*max(x, y)
```

[12]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a:str) -> 'a repeated 5 times'
```

Note that these annotations do **not** force a type on the parameters or the return value - they are simply there for documentation purposes within Python and **may** be used by external applications and modules, such as IDE's.

Just like docstrings are stored in the `__doc__` property, annotations are stored in the `__annotations__` property - a dictionary whose keys are the parameter names, and values are the annotation.

[13]:
```
my_func.__annotations__
```

[13]:
```
{'a': str, 'return': 'a repeated 5 times'}
```

Of course we can combine both docstrings and annotations:

[14]:
```
def fact(n: 'int >= 0')->int:
    '''Calculates n! (factorial function)

    Inputs:
        n: non-negative integer
    Returns:
        the factorial of n
    '''

    if n < 0:
```

```
        '''Note that this is not part of the docstring!'''
        return 1
    else:
        return n * fact(n-1)
```

[15]:
```
help(fact)
```

```
Help on function fact in module __main__:

fact(n:'int >= 0') -> int
    Calculates n! (factorial function)

    Inputs:
        n: non-negative integer
    Returns:
        the factorial of n
```

Annotations will work with default parameters too: just specify the default **after** the annotation:

[16]:
```
def my_func(a:str='a', b:int=1)->str:
    return a*b
```

[17]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a:str='a', b:int=1) -> str
```

[18]:
```
my_func()
```

[18]:
```
'a'
```

[19]:
```
my_func('abc', 3)
```

[19]:
```
'abcabcabc'
```

[20]:
```
def my_func(a:int=0, *args:'additional args'):
    print(a, args)
```

[21]:
```
my_func.__annotations__
```

[21]:
```
{'a': int, 'args': 'additional args'}
```

[22]:
```
help(my_func)
```

```
Help on function my_func in module __main__:

my_func(a:int=0, *args:'additional args')
```

0.0.2 Lambda Expressions

[1]:
```
lambda x: x**2
```

[1]:
```
<function __main__.<lambda>>
```

As you can see, the above expression just created a function.

Assigning to a Variable

[2]:
```
func = lambda x: x**2
```

[3]:
```
type(func)
```

[3]:
```
function
```

[4]:
```
func(3)
```

[4]:
```
9
```

We can specify arguments for lambdas just like we would for any function created using **def**, except for annotations:

[5]:
```
func_1 = lambda x, y=10: (x, y)
```

[6]:
```
func_1(1, 2)
```

[6]:
```
(1, 2)
```

[7]:
```
func_1(1)
```

[7]:
```
(1, 10)
```

We can even use * and **:

[8]:
```
func_2 = lambda x, *args, y, **kwargs: (x, *args, y, {**kwargs})
```

[9]:
```
func_2(1, 'a', 'b', y=100, a=10, b=20)
```

[9]:
```
(1, 'a', 'b', 100, {'a': 10, 'b': 20})
```

Passing as an Argument Lambdas are functions, and can therefore be passed to any other function as an argument (or returned from another function)

[10]:
```
def apply_func(x, fn):
    return fn(x)
```

[11]:
```
apply_func(3, lambda x: x**2)
```

[11]: 9

[12]:
```
apply_func(3, lambda x: x**3)
```

[12]: 27

Of course we can make this even more generic:

[13]:
```
def apply_func(fn, *args, **kwargs):
    return fn(*args, **kwargs)
```

[14]:
```
apply_func(lambda x, y: x+y, 1, 2)
```

[14]: 3

[15]:
```
apply_func(lambda x, *, y: x+y, 1, y=2)
```

[15]: 3

[16]:
```
apply_func(lambda *args: sum(args), 1, 2, 3, 4, 5)
```

[16]: 15

Of course in the example above, we really did not need to create a lambda!

[17]:
```
apply_func(sum, (1, 2, 3, 4, 5))
```

[17]: 15

Of course, we don't have to use lambdas when calling **apply_func**, we can also pass in a function defined using a **def** statement:

[18]:
```
def multiply(x, y):
    return x * y
```

[19]:
```
apply_func(multiply, 'a', 5)
```

[19]: 'aaaaa'

[20]:
```
apply_func(lambda x, y: x*y, 'a', 5)
```

[20]: 'aaaaa'

0.0.3 Lambdas and Sorting

Python has a built-in **sorted** method that can be used to sort any iterable. It will use the default ordering of the particular items, but sometimes you may want to (or need to) specify a different criteria for sorting.

Let's start with a simple list:

[1]:
```
l = ['a', 'B', 'c', 'D']
```

[2]:
```
sorted(l)
```

[2]:
```
['B', 'D', 'a', 'c']
```

As you can see there is a difference between upper and lower-case characters when sorting strings.

What if we wanted to make a case-insensitive sort?

Python's **sorted** function has a keyword-only argument that allows us to modify the values that are used to sort the list.

[3]:
```
sorted(l, key=str.upper)
```

[3]:
```
['a', 'B', 'c', 'D']
```

We could have used a lambda here (but you should not, this is just to illustrate using a lambda in this case):

[4]:
```
sorted(l, key = lambda s: s.upper())
```

[4]:
```
['a', 'B', 'c', 'D']
```

Let's look at how we might create a sorted list from a dictionary:

[5]:
```
d = {'def': 300, 'abc': 200, 'ghi': 100}
```

[6]:
```
d
```

[6]:
```
{'abc': 200, 'def': 300, 'ghi': 100}
```

[7]:
```
sorted(d)
```

[7]:
```
['abc', 'def', 'ghi']
```

What happened here?

Remember that iterating dictionaries actually iterates the keys - so we ended up with tyhe keys sorted alphabetically.

What if we want to return the keys sorted by their associated value instead?

[8]:
```
sorted(d, key=lambda k: d[k])
```

[8]:
```
['ghi', 'abc', 'def']
```

Maybe we want to sort complex numbers based on their distance from the origin:

[9]:
```
def dist(x):
    return (x.real)**2 + (x.imag)**2
```

[10]:
```
l = [3+3j, 1+1j, 0]
```

Trying to sort this list directly won't work since Python does not have an ordering defined for complex numbers:

[11]:
```
sorted(l)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-5ed0ddfda5a6> in <module>()
----> 1 sorted(l)

TypeError: '<' not supported between instances of 'complex' and 'complex'
```

Instead, let's try to specify the key using the distance:

[12]:
```
sorted(l, key=dist)
```

[12]:
```
[0, (1+1j), (3+3j)]
```

Of course, if we're only going to use the **dist** function once, we can just do the same thing this way:

[13]:
```
sorted(l, key=lambda x: (x.real)**2 + (x.imag)**2)
```

[13]:
```
[0, (1+1j), (3+3j)]
```

And here's another example where we want to sort a list of strings based on the **last character** of the string:

[14]:
```
l = ['Cleese', 'Idle', 'Palin', 'Chapman', 'Gilliam', 'Jones']
```

[15]:
```
sorted(l)
```

[15]:
```
['Chapman', 'Cleese', 'Gilliam', 'Idle', 'Jones', 'Palin']
```

[16]:
```
sorted(l, key=lambda s: s[-1])
```

[16]:
```
['Cleese', 'Idle', 'Gilliam', 'Palin', 'Chapman', 'Jones']
```

0.0.4 Challenge: Randomizing an Iterable using Sorted

[1]:
```
import random
```

[2]:
```
help(random.random)
```

```
Help on built-in function random:

random(…) method of random.Random instance
    random() -> x in the interval [0, 1).
```

[3]:
```
random.random()
```

[3]:
```
0.8655691916467607
```

[4]:
```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[5]:
```
sorted(l, key=lambda x: random.random())
```

[5]:
```
[5, 7, 2, 1, 3, 10, 9, 6, 8, 4]
```

Of course, this works for any iterable:

[6]:
```
sorted('abcdefg', key = lambda x: random.random())
```

[6]:
```
['b', 'd', 'g', 'e', 'a', 'c', 'f']
```

And to get a string back instead of just a list:

[7]:
```
''.join(sorted('abcdefg', key = lambda x: random.random()))
```

[7]:
```
'adfegbc'
```

0.0.5 Function Introspection

[1]:
```
def fact(n: "some non-negative integer") -> "n! or 0 if n < 0":
    """Calculates the factorial of a non-negative integer n

    If n is negative, returns 0.
    """
    if n < 0:
        return 0
    elif n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

Since functions are objects, we can add attributes to a function:

[2]:
```
fact.short_description = "factorial function"
```

[3]:
```
print(fact.short_description)
```

```
factorial function
```

We can see all the attributes that belong to a function using the **dir** function:

[4]:
```
dir(fact)
```

[4]:
```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__get__',
 '__getattribute__',
 '__globals__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__kwdefaults__',
 '__le__',
 '__lt__',
 '__module__',
 '__name__',
 '__ne__',
 '__new__',
 '__qualname__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'short_description']
```

We can see our **short_description** attribute, as well as some attributes we have seen before: **annotations** and **doc**:

[5]:
```
fact.__doc__
```

[5]:
```
'Calculates the factorial of a non-negative integer n\n    \n    If n is
negative, returns 0.\n    '
```

[6]:
```
fact.__annotations__
```

[6]:
```
{'n': 'some non-negative integer', 'return': 'n! or 0 if n < 0'}
```

We'll revisit some of these attributes later in this course, but let's take a look at a few here:

[7]:
```
def my_func(a, b=2, c=3, *, kw1, kw2=2, **kwargs):
    pass
```

Let's assign my_func to another variable:

[8]:
```
f = my_func
```

The **name** attribute holds the function's name:

[9]:
```
my_func.__name__
```

[9]:
```
'my_func'
```

[10]:
```
f.__name__
```

[10]:
```
'my_func'
```

The **defaults** attribute is a tuple containing any positional parameter defaults:

[11]:
```
my_func.__defaults__
```

[11]:
```
(2, 3)
```

[12]:
```
my_func.__kwdefaults__
```

[12]:
```
{'kw2': 2}
```

Let's create a function with some local variables:

[13]:
```
def my_func(a, b=1, *args, **kwargs):
    i = 10
    b = min(i, b)
    return a * b
```

[14]:
```
my_func('a', 100)
```

[14]:
```
'aaaaaaaaaa'
```

The **code** attribute contains a **code** object:

[15]:
```
my_func.__code__
```

[15]:
```
<code object my_func at 0x0000016640E71300, file "<ipython-
input-13-785cf1a800f4>", line 1>
```

This **code** object itself has various properties:

[16]:
```
dir(my_func.__code__)
```

[16]:
```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'co_argcount',
 'co_cellvars',
 'co_code',
 'co_consts',
 'co_filename',
 'co_firstlineno',
 'co_flags',
 'co_freevars',
 'co_kwonlyargcount',
 'co_lnotab',
 'co_name',
 'co_names',
```

```
 'co_nlocals',
 'co_stacksize',
 'co_varnames']
```

Attribute **co__varnames** is a tuple containing the parameter names and local variables:

[17]:
```
my_func.__code__.co_varnames
```

[17]:
```
('a', 'b', 'args', 'kwargs', 'i')
```

Attribute **co__argcount** returns the number of arguments (minus any * and ** args)

[18]:
```
my_func.__code__.co_argcount
```

[18]:
```
2
```

The inspect module It is much easier to use the **inspect** module!

[19]:
```
import inspect
```

[20]:
```
inspect.isfunction(my_func)
```

[20]:
```
True
```

By the way, there is a difference between a function and a method! A method is a function that is bound to some object:

[21]:
```
inspect.ismethod(my_func)
```

[21]:
```
False
```

[22]:
```
class MyClass:
    def f_instance(self):
        pass

    @classmethod
    def f_class(cls):
        pass

    @staticmethod
    def f_static():
        pass
```

Instance methods are bound to the **instance** of a class (not the class itself)

Class methods are bound to the **class**, not instances

Static methods are no bound either to the class or its instances

[23]:
```
inspect.isfunction(MyClass.f_instance), inspect.ismethod(MyClass.f_instance)
```

[23]:
```
(True, False)
```

[24]:
```
inspect.isfunction(MyClass.f_class), inspect.ismethod(MyClass.f_class)
```

[24]:
```
(False, True)
```

[25]:
```
inspect.isfunction(MyClass.f_static), inspect.ismethod(MyClass.f_static)
```

[25]:
```
(True, False)
```

[26]:
```
my_obj = MyClass()
```

[27]:
```
inspect.isfunction(my_obj.f_instance), inspect.ismethod(my_obj.f_instance)
```

[27]:
```
(False, True)
```

[28]:
```
inspect.isfunction(my_obj.f_class), inspect.ismethod(my_obj.f_class)
```

[28]:
```
(False, True)
```

[29]:
```
inspect.isfunction(my_obj.f_static), inspect.ismethod(my_obj.f_static)
```

[29]:
```
(True, False)
```

If you just want to know if something is a function or method:

[30]:
```
inspect.isroutine(my_func)
```

[30]:
```
True
```

[31]:
```
inspect.isroutine(MyClass.f_instance)
```

[31]:
```
True
```

[32]:
```
inspect.isroutine(my_obj.f_class)
```

[32]:
```
True
```

[33]:
```
inspect.isroutine(my_obj.f_static)
```

[33]:
```
True
```

We'll revisit this in more detail in section on OOP.

Introspecting Callable Code We can get back the source code of our function using the **getsource()** method:

[34]:
```
inspect.getsource(fact)
```

[34]:
```
'def fact(n: "some non-negative integer") -> "n! or 0 if n < 0":\n
"""Calculates the factorial of a non-negative integer n\n    \n    If n is
negative, returns 0.\n    """\n    if n <= 1:\n        return 1\n    else:\n
return n * fact(n-1)\n'
```

[35]:
```
print(inspect.getsource(fact))
```

```
def fact(n: "some non-negative integer") -> "n! or 0 if n < 0":
    """Calculates the factorial of a non-negative integer n

    If n is negative, returns 0.
    """
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

[36]:
```
inspect.getsource(MyClass.f_instance)
```

[36]:
```
'    def f_instance(self):\n        pass\n'
```

[37]:
```
inspect.getsource(my_obj.f_instance)
```

[37]:
```
'    def f_instance(self):\n        pass\n'
```

We can also find out where the function was defined:

[38]:
```
inspect.getmodule(fact)
```

[38]:
```
<module '__main__'>
```

[39]:
```
inspect.getmodule(print)
```

[39]:
```
<module 'builtins' (built-in)>
```

[40]:
```
import math
```

[41]:
```
inspect.getmodule(math.sin)
```

[41]:
```
<module 'math' (built-in)>
```

[42]:
```
# setting up variable
i = 10

# comment line 1
# comment line 2
def my_func(a, b=1):
    # comment inside my_func
    pass
```

[43]:
```
inspect.getcomments(my_func)
```

[43]:
```
'# comment line 1\n# comment line 2\n'
```

[44]:
```
print(inspect.getcomments(my_func))
```

```
# comment line 1
# comment line 2
```

Introspecting Callable Signatures

[45]:
```
# TODO: Provide implementation
def my_func(a: 'a string',
            b: int = 1,
            *args: 'additional positional args',
            kw1: 'first keyword-only arg',
            kw2: 'second keyword-only arg' = 10,
            **kwargs: 'additional keyword-only args') -> str:
    """does something
       or other"""
    pass
```

[46]:
```
inspect.signature(my_func)
```

[46]:
```
<Signature (a:'a string', b:int=1, *args:'additional positional args',
kw1:'first keyword-only arg', kw2:'second keyword-only arg'=10,
**kwargs:'additional keyword-only args') -> str>
```

[47]:
```
type(inspect.signature(my_func))
```

[47]:
```
inspect.Signature
```

[48]:
```
sig = inspect.signature(my_func)
```

[49]:
```
dir(sig)
```

[49]:
```
['__class__',
 '__delattr__',
```

```
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setstate__',
 '__sizeof__',
 '__slots__',
 '__str__',
 '__subclasshook__',
 '_bind',
 '_bound_arguments_cls',
 '_hash_basis',
 '_parameter_cls',
 '_parameters',
 '_return_annotation',
 'bind',
 'bind_partial',
 'empty',
 'from_builtin',
 'from_callable',
 'from_function',
 'parameters',
 'replace',
 'return_annotation']
```

[50]:
```
for param_name, param in sig.parameters.items():
    print(param_name, param)
```

```
a a:'a string'
b b:int=1
args *args:'additional positional args'
kw1 kw1:'first keyword-only arg'
```

```
kw2 kw2:'second keyword-only arg'=10
kwargs **kwargs:'additional keyword-only args'
```

[51]:
```
def print_info(f: "callable") -> None:
    print(f.__name__)
    print('=' * len(f.__name__), end='\n\n')

    print('{0}\n{1}\n'.format(inspect.getcomments(f),
                              inspect.cleandoc(f.__doc__)))

    print('{0}\n{1}'.format('Inputs', '-'*len('Inputs')))

    sig = inspect.signature(f)
    for param in sig.parameters.values():
        print('Name:', param.name)
        print('Default:', param.default)
        print('Annotation:', param.annotation)
        print('Kind:', param.kind)
        print('-------------------------\n')

    print('{0}\n{1}'.format('\n\nOutput', '-'*len('Output')))
    print(sig.return_annotation)
```

[52]:
```
print_info(my_func)
```

```
my_func
=======

# TODO: Provide implementation

does something
or other

Inputs
------
Name: a
Default: <class 'inspect._empty'>
Annotation: a string
Kind: POSITIONAL_OR_KEYWORD
-------------------------

Name: b
Default: 1
Annotation: <class 'int'>
Kind: POSITIONAL_OR_KEYWORD
-------------------------

Name: args
```

```
Default: <class 'inspect._empty'>
Annotation: additional positional args
Kind: VAR_POSITIONAL
--------------------------

Name: kw1
Default: <class 'inspect._empty'>
Annotation: first keyword-only arg
Kind: KEYWORD_ONLY
--------------------------

Name: kw2
Default: 10
Annotation: second keyword-only arg
Kind: KEYWORD_ONLY
--------------------------

Name: kwargs
Default: <class 'inspect._empty'>
Annotation: additional keyword-only args
Kind: VAR_KEYWORD
--------------------------



Output
------
<class 'str'>
```

A Side Note on Positional Only Arguments Some built-in callables have arguments that are positional only (i.e. cannot be specified using a keyword).

However, Python does not currently have any syntax that allows us to define callables with positional only arguments.

In general, the documentation uses a / character to indicate that all preceding arguments are positional-only. But not always :-(

[53]:
```
help(divmod)
```

```
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

Here we see that the **divmod** function takes two positional-only parameters:

[54]:
```
divmod(10, 3)
```

[54]:
```
(3, 1)
```

[55]:
```
divmod(x=10, y=3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-c637b01eef33> in <module>()
----> 1 divmod(x=10, y=3)

TypeError: divmod() takes no keyword arguments
```

Similarly, the string **replace** function also takes positional-only arguments, however, the documentation does not indicate this!

[56]:
```
help(str.replace)
```

```
Help on method_descriptor:

replace(…)
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

[57]:
```
'abcdefg'.replace('abc', 'xyz')
```

[57]:
```
'xyzdefg'
```

[58]:
```
'abcdefg'.replace(old='abc', new='xyz')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-9d61ac657cae> in <module>()
----> 1 'abcdefg'.replace(old='abc', new='xyz')

TypeError: replace() takes no keyword arguments
```

0.0.6 Callables

A callable is an object that can be called (using the **()** operator), and always returns a value.

We can check if an object is callable by using the built-in function **callable**

Functions and Methods are callable

[1]:
```
callable(print)
```

[1]: True

[2]:
```
callable(len)
```

[2]: True

[3]:
```
l = [1, 2, 3]
callable(l.append)
```

[3]: True

[4]:
```
s = 'abc'
callable(s.upper)
```

[4]: True

Callables always return a value:

[5]:
```
result = print('hello')
print(result)
```

```
hello
None
```

[6]:
```
l = [1, 2, 3]
result = l.append(4)
print(result)
print(l)
```

```
None
[1, 2, 3, 4]
```

[7]:
```
s = 'abc'
result = s.upper()
print(result)
```

```
ABC
```

Classes are callable:

[8]:
```
from decimal import Decimal
```

[9]:
```
callable(Decimal)
```

[9]: True

[10]:
```
result = Decimal('10.5')
print(result)
```

```
10.5
```

Class instances may be callable:

[11]:
```
class MyClass:
    def __init__(self):
        print('initializing...')
        self.counter = 0

    def __call__(self, x=1):
        self.counter += x
        print(self.counter)
```

[12]:
```
my_obj = MyClass()
```

```
initializing...
```

[13]:
```
callable(my_obj.__init__)
```

[13]:
```
True
```

[14]:
```
callable(my_obj.__call__)
```

[14]:
```
True
```

[15]:
```
my_obj()
```

```
1
```

[16]:
```
my_obj()
```

```
2
```

[17]:
```
my_obj(10)
```

```
12
```

0.0.7 Higher-Order Functions: Map and Filter

Definition: A function that takes a function as an argument, and/or returns a function as its return value

For example, the **sorted** function is a higher-order function as we saw in an earlier video.

Map The **map** built-in function is a higher-order function that applies a function to an iterable type object:

[1]:
```
help(map)
```

```
Help on class map in module builtins:

class map(object)
 |  map(func, *iterables) --> map object
 |
 |  Make an iterator that computes the function using arguments from
 |  each of the iterables.  Stops when the shortest iterable is exhausted.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(…)
 |      Return state information for pickling.
```

[2]:
```
def fact(n):
    return 1 if n < 2 else n * fact(n-1)
```

[3]:
```
fact(3)
```

[3]: 6

[4]:
```
fact(4)
```

[4]: 24

[5]:
```
map(fact, [1, 2, 3, 4, 5])
```

[5]: <map at 0x23b123a3978>

The **map** function returns a **map** object, which is an **iterable** - we can either convert that to a list or enumerate it:

[6]:
```
l = list(map(fact, [1, 2, 3, 4, 5]))
print(l)
```

```
[1, 2, 6, 24, 120]
```

We can also use it this way:

[7]:
```
l1 = [1, 2, 3, 4, 5]
l2 = [10, 20, 30, 40, 50]

f = lambda x, y: x+y

m = map(f, l1, l2)
list(m)
```

[7]:
```
[11, 22, 33, 44, 55]
```

Filter

[8]:
```
help(filter)
```

```
Help on class filter in module builtins:

class filter(object)
 |  filter(function or None, iterable) --> filter object
 |
 |  Return an iterator yielding those items of iterable for which function(item)
 |  is true. If function is None, return the items that are true.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(…)
 |      Return state information for pickling.
```

The **filter** function is a function that filters an iterable based on the truthyness of the elements, or the truthyness of the elements after applying a function to them. Like the **map** function, the **filter** function returns an iterable that we can view by generating a list from it, or simply enumerating in a for loop.

[9]:
```
l = [0, 1, 2, 3, 4, 5, 6]
for e in filter(None, l):
    print(e)
```

```
1
2
3
4
5
6
```

Notice how **0** was eliminated from the list, since **0** is **falsy**.

We can use a function for this filtering.

Suppose we want to filter out all odd values, only retaining even values:

We could first define a function to return True if the value is even, and False otherwise:

[10]:
```
def is_even(n):
    return n % 2 == 0
```

[11]:
```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
result = filter(is_even, l)
print(list(result))
```

```
[2, 4, 6, 8]
```

Of course, we could just use a lambda expression instead:

[12]:
```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
result = filter(lambda x: x % 2 == 0, l)
print(list(result))
```

```
[2, 4, 6, 8]
```

Alternatives to map and filter using Comprehensions We'll cover comprehensions in much more detail later, but, for now, just be aware that we can use comprehensions instead of the **map** and **filter** functions - you decide which one you find more readable and enjoyable to write.

Map using a list comprehension:

- factorial example

[13]:
```
l = [1, 2, 3, 4, 5]
result = [fact(i) for i in l]
print(result)
```

```
[1, 2, 6, 24, 120]
```

- two iterables example

Before we do this example we need to know about the **zip** function.

The **zip** built-in function will take one or more iterables, and generate an iterable of tuples where each tuple contains one element from each iterable:

[14]:
```
l1 = 1, 2, 3
l2 = 'a', 'b', 'c'
list(zip(l1, l2))
```

[14]:
```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

[15]:
```
l1 = 1, 2, 3
l2 = [10, 20, 30]
l3 = ('a', 'b', 'c')
list(zip(l1, l2, l3))
```

[15]:
```
[(1, 10, 'a'), (2, 20, 'b'), (3, 30, 'c')]
```

[5]:
```
l1 = [1, 2, 3]
l2 = (10, 20, 30)
l3 = 'abc'
list(zip(l1, l2, l3))
```

[5]:
```
[(1, 10, 'a'), (2, 20, 'b'), (3, 30, 'c')]
```

[7]:
```
l1 = range(100)
l2 = 'python'
list(zip(l1, l2))
```

[7]:
```
[(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

Using the **zip** function we can now add our two lists element by element as follows:

[16]:
```
l1 = [1, 2, 3, 4, 5]
l2 = [10, 20, 30, 40, 50]
result = [i + j for i,j in zip(l1,l2)]
print(result)
```

```
[11, 22, 33, 44, 55]
```

Filtering using a comprehension We can very easily filter an iterable using a comprehension as follows:

[17]:
```
l = [1, 2, 3, 4, 5, 6, 7, 8, 9]

result = [i for i in l if i % 2 == 0]
print(result)
```

```
[2, 4, 6, 8]
```

As you can see, we did not even need a lambda expression!

Combining map and filter

[1]:
```
list(filter(lambda y: y < 25, map(lambda x: x**2, range(10))))
```

[1]:
```
[0, 1, 4, 9, 16]
```

Alternatively, we can use a list comprehension to do the same thing:

[2]:
```
[x**2 for x in range(10) if x**2 < 25]
```

[2]:
```
[0, 1, 4, 9, 16]
```

We will come back, in more detail, to comprehensions and generators later in this course.

0.0.8 Reducing Functions in Python

Maximum and Minimum Suppose we want to find the maximum value in a list:

[1]:
```
l = [5, 8, 6, 10, 9]
```

We can solve this problem using a **for** loop.

First we define a function that returns the maximum of two arguments:

[2]:
```
_max = lambda a, b: a if a > b else b
```

[3]:
```
def max_sequence(sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = _max(result, x)
    return result
```

[4]:
```
max_sequence(l)
```

[4]:
```
10
```

To calculate the minimum, all we need to do is to change the function that is repeatedly applied:

[5]:
```
_min = lambda a, b: a if a < b else b
```

[6]:
```
def min_sequence(sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = _min(result, x)
    return result
```

[7]:
```
print(l)
print(min_sequence(l))
```

```
[5, 8, 6, 10, 9]
5
```

In general we could write it like this:

[8]:
```
def _reduce(fn, sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = fn(result, x)
    return result
```

[9]:
```
_reduce(_max, l)
```

[9]:
```
10
```

[10]:
```
_reduce(_min, l)
```

[10]:
```
5
```

We could even just use a lambda directly in the call to **__reduce**:

[11]:
```
_reduce(lambda a, b: a if a > b else b, l)
```

[11]:
```
10
```

[12]:
```
_reduce(lambda a, b: a if a < b else b, l)
```

[12]:
```
5
```

Using the same approach, we could even add all the elements of a sequence together:

[13]:
```
print(l)
```

```
[5, 8, 6, 10, 9]
```

[14]:
```
_reduce(lambda a, b: a + b, l)
```

[14]:
```
38
```

Python actually implements a reduce function, which is found in the **functools** module. Unlike our **__reduce** function, it can handle any iterable, not just sequences.

[15]:
```
from functools import reduce
```

[16]:
```
l
```

[16]:
```
[5, 8, 6, 10, 9]
```

[17]:
```
reduce(lambda a, b: a if a > b else b, l)
```

[17]:
```
10
```

[18]:
```
reduce(lambda a, b: a if a < b else b, l)
```

[18]:
```
5
```

[19]:
```
reduce(lambda a, b: a + b, l)
```

[19]:
```
38
```

Finding the max and min of an iterable is such a common thing that Python provides a built-in function to do just that:

[20]:
```
max(l), min(l)
```

[20]:
```
(10, 5)
```

Finding the sum of all the elements in an iterable is also common enough that Python implements the **sum** function:

[21]:
```
sum(l)
```

[21]:
```
38
```

The any and all built-ins Python provides two additional built-in reducing functions: **any** and **all**.

The **any** function will return **True** if any element in the iterable is truthy:

[22]:
```
l = [0, 1, 2]
any(l)
```

[22]:
```
True
```

[23]:
```
l = [0, 0, 0]
any(l)
```

[23]:
```
False
```

On the other hand, **all** will return True if **every** element of the iterable is truthy:

[24]:
```
l = [0, 1, 2]
all(l)
```

[24]:
```
False
```

[25]:
```
l = [1, 2, 3]
all(l)
```

[25]: True

We can implement these functions ourselves using **reduce** if we choose to - simply use the Boolean **or** or **and** operators as the function passed to **reduce** to implement **any** and **all** respectively.

any

[26]:
```
l = [0, 1, 2]
reduce(lambda a, b: bool(a or b), l)
```

[26]: True

[27]:
```
l = [0, 0, 0]
reduce(lambda a, b: bool(a or b), l)
```

[27]: False

all

[28]:
```
l = [0, 1, 2]
reduce(lambda a, b: bool(a and b), l)
```

[28]: False

[29]:
```
l = [1, 2, 3]
reduce(lambda a, b: bool(a and b), l)
```

[29]: True

Products Sometimes we may want to find the product of every element of an iterable.

Python does not provide us a built-in method to do this, so we have to either use a procedural approach, or we can use the **reduce** function.

We start by defining a function that multiplies two arguments together:

[30]:
```
def mult(a, b):
    return a * b
```

Then we can use the **reduce** function:

[31]:
```
l = [2, 3, 4]
reduce(mult, l)
```

[31]: 24

Remember what this did:

```
step 1: result = 2
step 2: result = mult(result, 3) = mult(2, 3) = 6
```

```
step 3: result = mult(result, 4) = mult(6, 4) = 24
step 4: l exhausted, return result --> 24
```

Of course, we can also just use a lambda:

[32]:
```
reduce(lambda a, b: a * b, l)
```

[32]:
```
24
```

Factorials

Factorials A special case of the product we just did would be calculating the factorial of some number (**n!**):

Recall:

```
n! = 1 * 2 * 3 * ... * n
```

In other words, we are calculating the product of a sequence containing consecutive integers from 1 to n (inclusive)

We can easily write this using a simple for loop:

[33]:
```
def fact(n):
    if n <= 1:
        return 1
    else:
        result = 1
        for i in range(2, n+1):
            result *= i
        return result
```

[34]:
```
fact(1), fact(2), fact(3), fact(4), fact(5)
```

[34]:
```
(1, 2, 6, 24, 120)
```

We could also write this using a recursive function:

[35]:
```
def fact(n):
    if n <=1:
        return 1
    else:
        return n * fact(n-1)
```

[36]:
```
fact(1), fact(2), fact(3), fact(4), fact(5)
```

[36]:
```
(1, 2, 6, 24, 120)
```

Finally we can also write this using **reduce** as follows:

[37]:
```
n = 5
reduce(lambda a, b: a * b, range(1, n+1))
```

[37]:
```
120
```

As you can see, the **reduce** approach, although concise, is sometimes more difficult to understand than the plain loop or recursive approach.

reduce initializer Suppose we want to provide some sort of default when we claculate the product of the elements of an iterable if that iterable is empty:

[38]:
```
l = [1, 2, 3]
reduce(lambda x, y: x*y, l)
```

[38]:
```
6
```

but if **l** is empty:

[39]:
```
l = []
reduce(lambda x, y: x*y, l)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-09fa1e2b48dc> in <module>()
      1 l = []
----> 2 reduce(lambda x, y: x*y, l)

TypeError: reduce() of empty sequence with no initial value
```

To fix this, we can provide an initializer. In this case, we will use **1** since that will not affect the result of the product, and still allow us to return a value for an empty iterable.

[40]:
```
l = []
reduce(lambda x, y: x*y, l, 1)
```

[40]:
```
1
```

[ ]:
```

```

0.0.9 Partial Functions

[1]:
```
from functools import partial
```

[2]:
```
def my_func(a, b, c):
    print(a, b, c)
```

[3]:
```
f = partial(my_func, 10)
```

[4]:
```
f(20, 30)
```

```
10 20 30
```

We could have done this using another function (or a lambda) as well:

[5]:
```
def partial_func(b, c):
    return my_func(10, b, c)
```

[6]:
```
partial_func(20, 30)
```

```
10 20 30
```

or, using a lambda:

[7]:
```
fn = lambda b, c: my_func(10, b, c)
```

[8]:
```
fn(20, 30)
```

```
10 20 30
```

Any of these ways is fine, but sometimes partial is just a cleaner more consise way to do it.

Also, it is quite flexible with parameters:

[9]:
```
def my_func(a, b, *args, k1, k2, **kwargs):
    print(a, b, args, k1, k2, kwargs)
```

[10]:
```
f = partial(my_func, 10, k1='a')
```

[11]:
```
f(20, 30, 40, k2='b', k3='c')
```

```
10 20 (30, 40) a b {'k3': 'c'}
```

We can of course do the same thing using a regular function too:

[12]:
```
def f(b, *args, k2, **kwargs):
    return my_func(10, b, *args, k1='a', k2=k2, **kwargs)
```

[13]:
```
f(20, 30, 40, k2='b', k3='c')
```

```
10 20 (30, 40) a b {'k3': 'c'}
```

As you can see in this case, using **partial** seems a lot simpler.

Also, you are not stuck having to specify the first argument in your partial:

[14]:
```
def power(base, exponent):
    return base ** exponent
```

[15]:
```
power(2, 3)
```

[15]:
```
8
```

[16]:
```
square = partial(power, exponent=2)
```

[17]:
```
square(4)
```

[17]:
```
16
```

[18]:
```
cube = partial(power, exponent=3)
```

[19]:
```
cube(2)
```

[19]:
```
8
```

You can even call it this way:

[20]:
```
cube(base=3)
```

[20]:
```
27
```

Caveat We can certainly use variables instead of literals when creating partials, but we have to be careful.

[21]:
```
def my_func(a, b, c):
    print(a, b, c)
```

[22]:
```
a = 10
f = partial(my_func, a)
```

[23]:
```
f(20, 30)
```

```
10 20 30
```

Now let's change the value of the variable **a** and see what happens:

[24]:
```
a = 100
```

[25]:
```
f(20, 30)
```

```
10 20 30
```

As you can see, the value for **a** is fixed once the partial has been created.

In fact, the memory address of **a** is baked in to the partial, and **a** is immutable.

If we use a mutable object, things are different:

[26]:
```
a = [10, 20]
f = partial(my_func, a)
```

[27]:
```
f(100, 200)
```

```
[10, 20] 100 200
```

[28]:
```
a.append(30)
```

[29]:
```
f(100, 200)
```

```
[10, 20, 30] 100 200
```

Use Cases We tend to use partials in situation where we need to call a function that actually requires more parameters than we can supply.

Often this is because we are working with exiting libraries or code, and we have a special case.

For example, suppose we have points (represented as tuples), and we want to sort them based on the distance of the point from some other fixed point:

[30]:
```
origin = (0, 0)
```

[31]:
```
l = [(1,1), (0, 2), (-3, 2), (0,0), (10, 10)]
```

[32]:
```
dist2 = lambda x, y: (x[0]-y[0])**2 + (x[1]-y[1])**2
```

[33]:
```
dist2((0,0), (1,1))
```

[33]:
```
2
```

[34]:
```
sorted(l, key = lambda x: dist2((0,0), x))
```

[34]:
```
[(0, 0), (1, 1), (0, 2), (-3, 2), (10, 10)]
```

[35]:
```
sorted(l, key=partial(dist2, (0,0)))
```

[35]:
```
[(0, 0), (1, 1), (0, 2), (-3, 2), (10, 10)]
```

Another use case is when using **callback** functions. Usually these are used when running asynchronous operations, and you provide a callable to another callable which will be called when the first callable completes its execution.

Very often, the asynchronous callable will specify the number of variables that the callback function must have - this may not be what we want, maybe we want to add some additional info.

We'll look at asynchronous processing later in this course.

Often we can also use partial functions to make our life a bit easier.

Consider a situation where we have some generic `email()` function that can be used to notify someone when various things happen in our application. But depending on what is happening we may want to notify different people. Let's see how we may do this:

[36]:
```
def sendmail(to, subject, body):
    # code to send email
    print('To:{0}, Subject:{1}, Body:{2}'.format(to, subject, body))
```

Now, we may haver different email adresses we want to send notifications to, maybe defined in a config file in our app. Here, I'll just use hardcoded variables:

[37]:
```
email_admin = 'palin@python.edu'
email_devteam = 'idle@python.edu;cleese@python.edu'
```

Now when we want to send emails we would have to write things like:

[38]:
```
sendmail(email_admin, 'My App Notification', 'the parrot is dead.')
sendmail(';'.join((email_admin, email_devteam)), 'My App Notification', 'the␣
 ↪ministry is closed until further notice.')
```

```
To:palin@python.edu, Subject:My App Notification, Body:the parrot is dead.
To:palin@python.edu;idle@python.edu;cleese@python.edu, Subject:My App
Notification, Body:the ministry is closed until further notice.
```

We could simply our life a little using partials this way:

[39]:
```
send_admin = partial(sendmail, email_admin, 'For you eyes only')
send_dev = partial(sendmail, email_devteam, 'Dear IT:')
send_all = partial(sendmail, ';'.join((email_admin, email_devteam)), 'Loyal␣
 ↪Subjects')
```

[40]:
```
send_admin('the parrot is dead.')
send_all('the ministry is closed until further notice.')
```

```
To:palin@python.edu, Subject:For you eyes only, Body:the parrot is dead.
To:palin@python.edu;idle@python.edu;cleese@python.edu, Subject:Loyal Subjects,
Body:the ministry is closed until further notice.
```

Finally, let's make this a little more complex, with a mixture of positional and keyword-only arguments:

[41]:
```
def sendmail(to, subject, body, *, cc=None, bcc=email_devteam):
    # code to send email
    print('To:{0}, Subject:{1}, Body:{2}, CC:{3}, BCC:{4}'.format(to,
                                                                  subject,
                                                                  body,
                                                                  cc,
                                                                  bcc))
```

[42]:
```
send_admin = partial(sendmail, email_admin, 'General Admin')
send_admin_secret = partial(sendmail, email_admin, 'For your eyes only',␣
 ↪cc=None, bcc=None)
```

[43]:
```
send_admin('and now for something completely different')
```

```
To:palin@python.edu, Subject:General Admin, Body:and now for something
completely different, CC:None, BCC:idle@python.edu;cleese@python.edu
```

[44]:
```
send_admin_secret('the parrot is dead!')
```

```
To:palin@python.edu, Subject:For your eyes only, Body:the parrot is dead!,
CC:None, BCC:None
```

[45]:
```
send_admin_secret('the parrot is no more!', bcc=email_devteam)
```

```
To:palin@python.edu, Subject:For your eyes only, Body:the parrot is no more!,
CC:None, BCC:idle@python.edu;cleese@python.edu
```

[49]:
```
def pow(base, exponent):
    return base ** exponent
```

[52]:
```
cube = partial(pow, exponent=3)
```

[53]:
```
cube(2)
```

[53]: 8

[54]:
```
cube(2, 4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-725d549b8104> in <module>()
----> 1 cube(2, 4)

TypeError: pow() got multiple values for argument 'exponent'
```

[55]:
```
cube(2, exponent=4)
```

[55]: 16

[ ]:
```

```

0.0.10 The operator Module

[1]:
```
import operator
```

[2]:
```
dir(operator)
```

```
[2]: ['__abs__',
      '__add__',
      '__all__',
      '__and__',
      '__builtins__',
      '__cached__',
      '__concat__',
      '__contains__',
      '__delitem__',
      '__doc__',
      '__eq__',
      '__file__',
      '__floordiv__',
      '__ge__',
      '__getitem__',
      '__gt__',
      '__iadd__',
      '__iand__',
      '__iconcat__',
      '__ifloordiv__',
      '__ilshift__',
      '__imatmul__',
      '__imod__',
      '__imul__',
      '__index__',
      '__inv__',
      '__invert__',
      '__ior__',
      '__ipow__',
      '__irshift__',
      '__isub__',
      '__itruediv__',
      '__ixor__',
      '__le__',
      '__loader__',
      '__lshift__',
      '__lt__',
      '__matmul__',
      '__mod__',
      '__mul__',
      '__name__',
      '__ne__',
      '__neg__',
      '__not__',
      '__or__',
      '__package__',
      '__pos__',
```

```
'__pow__',
'__rshift__',
'__setitem__',
'__spec__',
'__sub__',
'__truediv__',
'__xor__',
'_abs',
'abs',
'add',
'and_',
'attrgetter',
'concat',
'contains',
'countOf',
'delitem',
'eq',
'floordiv',
'ge',
'getitem',
'gt',
'iadd',
'iand',
'iconcat',
'ifloordiv',
'ilshift',
'imatmul',
'imod',
'imul',
'index',
'indexOf',
'inv',
'invert',
'ior',
'ipow',
'irshift',
'is_',
'is_not',
'isub',
'itemgetter',
'itruediv',
'ixor',
'le',
'length_hint',
'lshift',
'lt',
'matmul',
```

```
 'methodcaller',
 'mod',
 'mul',
 'ne',
 'neg',
 'not_',
 'or_',
 'pos',
 'pow',
 'rshift',
 'setitem',
 'sub',
 'truediv',
 'truth',
 'xor']
```

Arithmetic Operators A variety of arithmetic operators are implemented.

[3]:
```
operator.add(1, 2)
```

[3]: 3

[4]:
```
operator.mul(2, 3)
```

[4]: 6

[5]:
```
operator.pow(2, 3)
```

[5]: 8

[6]:
```
operator.mod(13, 2)
```

[6]: 1

[7]:
```
operator.floordiv(13, 2)
```

[7]: 6

[8]:
```
operator.truediv(3, 2)
```

[8]: 1.5

These would have been very handy in our previous section:

[9]:
```
from functools import reduce
```

[10]:
```
reduce(lambda x, y: x*y, [1, 2, 3, 4])
```

[10]:
```
24
```

Instead of defining a lambda, we could simply use **operator.mul**:

[11]:
```
reduce(operator.mul, [1, 2, 3, 4])
```

[11]:
```
24
```

Comparison and Boolean Operators Comparison and Boolean operators are also implemented as functions:

[12]:
```
operator.lt(10, 100)
```

[12]:
```
True
```

[13]:
```
operator.le(10, 10)
```

[13]:
```
True
```

[14]:
```
operator.is_('abc', 'def')
```

[14]:
```
False
```

We can even get the truthyness of an object:

[15]:
```
operator.truth([1,2])
```

[15]:
```
True
```

[16]:
```
operator.truth([])
```

[16]:
```
False
```

[17]:
```
operator.and_(True, False)
```

[17]:
```
False
```

[18]:
```
operator.or_(True, False)
```

[18]:
```
True
```

Element and Attribute Getters and Setters We generally select an item by index from a sequence by using **[n]**:

[19]:
```
my_list = [1, 2, 3, 4]
my_list[1]
```

[19]:
```
2
```

We can do the same thing using:

[20]:
```
operator.getitem(my_list, 1)
```

[20]:
```
2
```

If the sequence is mutable, we can also set or remove items:

[21]:
```
my_list = [1, 2, 3, 4]
my_list[1] = 100
del my_list[3]
print(my_list)
```

```
[1, 100, 3]
```

[22]:
```
my_list = [1, 2, 3, 4]
operator.setitem(my_list, 1, 100)
operator.delitem(my_list, 3)
print(my_list)
```

```
[1, 100, 3]
```

We can also do the same thing using the **operator** module's **itemgetter** function.

The difference is that this returns a callable:

[23]:
```
f = operator.itemgetter(2)
```

Now, **f(my_list)** will return **my_list[2]**

[24]:
```
f(my_list)
```

[24]:
```
3
```

[25]:
```
x = 'python'
f(x)
```

[25]:
```
't'
```

Furthermore, we can pass more than one index to **itemgetter**:

[26]:
```
f = operator.itemgetter(2, 3)
```

[27]:
```
my_list = [1, 2, 3, 4]
f(my_list)
```

[27]:
```
(3, 4)
```

[28]:
```
x = 'pytyhon'
f(x)
```

[28]:
```
('t', 'y')
```

Similarly, **operator.attrgetter** does the same thing, but with object attributes.

[29]:
```
class MyClass:
    def __init__(self):
        self.a = 10
        self.b = 20
        self.c = 30

    def test(self):
        print('test method running...')
```

[30]:
```
obj = MyClass()
```

[31]:
```
obj.a, obj.b, obj.c
```

[31]:
```
(10, 20, 30)
```

[32]:
```
f = operator.attrgetter('a')
```

[33]:
```
f(obj)
```

[33]:
```
10
```

[34]:
```
my_var = 'b'
operator.attrgetter(my_var)(obj)
```

[34]:
```
20
```

[35]:
```
my_var = 'c'
operator.attrgetter(my_var)(obj)
```

[35]:
```
30
```

[36]:
```
f = operator.attrgetter('a', 'b', 'c')
```

[37]:
```
f(obj)
```

[37]:
```
(10, 20, 30)
```

Of course, attributes can also be methods.

In this case, **attrgetter** will return the object's **test** method - a callable that can then be called using **()**:

[38]:
```
f = operator.attrgetter('test')
```

[39]:
```
obj_test_method = f(obj)
```

[40]:
```
obj_test_method()
```

```
test method running…
```

Just like lambdas, we do not need to assign them to a variable name in order to use them:

[41]:
```
operator.attrgetter('a', 'b')(obj)
```

[41]:
```
(10, 20)
```

[42]:
```
operator.itemgetter(2, 3)('python')
```

[42]:
```
('t', 'h')
```

Of course, we can achieve the same thing using functions or lambdas:

[43]:
```
f = lambda x: (x.a, x.b, x.c)
```

[44]:
```
f(obj)
```

[44]:
```
(10, 20, 30)
```

[45]:
```
f = lambda x: (x[2], x[3])
```

[46]:
```
f([1, 2, 3, 4])
```

[46]:
```
(3, 4)
```

[47]:
```
f('python')
```

[47]:
```
('t', 'h')
```

Use Case Example: Sorting Suppose we want to sort a list of complex numbers based on the real part of the numbers:

[48]:
```
a = 2 + 5j
a.real
```

[48]:
```
2.0
```

[49]:
```
l = [10+1j, 8+2j, 5+3j]
sorted(l, key=operator.attrgetter('real'))
```

[49]:
```
[(5+3j), (8+2j), (10+1j)]
```

Or if we want to sort a list of string based on the last character of the strings:

[50]:
```
l = ['aaz', 'aad', 'aaa', 'aac']
sorted(l, key=operator.itemgetter(-1))
```

[50]:
```
['aaa', 'aac', 'aad', 'aaz']
```

Or maybe we want to sort a list of tuples based on the first item of each tuple:

[51]:
```
l = [(2, 3, 4), (1, 2, 3), (4, ), (3, 4)]
sorted(l, key=operator.itemgetter(0))
```

[51]:
```
[(1, 2, 3), (2, 3, 4), (3, 4), (4,)]
```

Slicing

[52]:
```
l = [1, 2, 3, 4]
```

[53]:
```
l[0:2]
```

[53]:
```
[1, 2]
```

[54]:
```
l[0:2] = ['a', 'b', 'c']
print(l)
```

```
['a', 'b', 'c', 3, 4]
```

[55]:
```
del l[3:5]
print(l)
```

```
['a', 'b', 'c']
```

We can do the same thing this way:

[56]:
```
l = [1, 2, 3, 4]
```

[57]:
```
operator.getitem(l, slice(0,2))
```

[57]:
```
[1, 2]
```

[58]:
```
operator.setitem(l, slice(0,2), ['a', 'b', 'c'])
print(l)
```

```
['a', 'b', 'c', 3, 4]
```

[59]:
```
operator.delitem(l, slice(3, 5))
print(l)
```

```
['a', 'b', 'c']
```

Calling another Callable

[60]:
```
x = 'python'
x.upper()
```

[60]:
```
'PYTHON'
```

[61]:
```
operator.methodcaller('upper')('python')
```

[61]:
```
'PYTHON'
```

Of course, since **upper** is just an attribute of the string object **x**, we could also have used:

[62]:
```
operator.attrgetter('upper')(x)()
```

[62]:
```
'PYTHON'
```

If the callable takes in more than one parameter, they can be specified as additional arguments in **methodcaller**:

[63]:
```
class MyClass:
    def __init__(self):
        self.a = 10
        self.b = 20

    def do_something(self, c):
        print(self.a, self.b, c)
```

[64]:
```
obj = MyClass()
```

[65]:
```
obj.do_something(100)
```

```
10 20 100
```

[66]:
```
operator.methodcaller('do_something', 100)(obj)
```

```
10 20 100
```

[67]:
```
class MyClass:
    def __init__(self):
        self.a = 10
        self.b = 20

    def do_something(self, *, c):
        print(self.a, self.b, c)
```

[68]:
```
obj.do_something(c=100)
```

```
10 20 100
```

[69]:
```
operator.methodcaller('do_something', c=100)(obj)
```

```
10 20 100
```

More information on the **operator** module can be found here:

https://docs.python.org/3/library/operator.html