01-modules-namespaces

March 21, 2024

1 Modules and Namespaces

1.0.1 Fundamental Observations

The built-in python function `dir()` provides a mechanism that asks, "what is available here in this particular scope?"

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

```
['In', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__', '__vsc_ipynb_file__',
'_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh', 'exit', 'get_ipython', 'open',
'quit']
```

Normally we would see something shorter like `['__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'sys']` but this is a python notebook so there are some more options.

Either way, these identifiers represent things accessible in the current scope, which means we can look at both their values and types.

Generally when not in a notebook `__builtins__` is a dictionary that represents the names of things that are built into Python (usable without importing any modules). Because it acts like a dictionary, we can use it as such e.g. `__builtins__['list'](stuff)` corresponds to `list(stuff)`. We could also technically manipulate this dictionary to redefine what is "built in" to Python, although this is generally not advisable, just something to keep in mind in terms of how Python works.

1.0.2 Scopes, namespaces, functions

Now suppose we declare a variable, how does the output of `dir()` change?

[4]:
```
'abc' in dir()
```

[4]:
```
False
```

[5]:
```
hello = 'world'
'hello' in dir()
```

[5]:
```
True
```

[6]:
```
del hello
'hello' in dir()
```

[6]:
```
False
```

These observations seem to suggest that the creation of an identifier adds it to some directory, and deleting it removes it as such. The creation of functions is also the same as the `def` statement is equivalent to declaring some variable, and storing a function object within it.

1.0.3 LEGB

LEGB is a rule that represents how identifier resolution occurs in Python. LEGB: Local, Enclosed, Global, Builtins. When we utilize any identifier in Python, the way it is *resolved* is through looking these scopes in the order of L -> E -> G -> B.

The `__builtins__` method from earlier seems to show us what exists in builtins. We can also use `locals()` and `globals()` to see what exists in those scopes as well.

[8]:
```
globals().keys()
```

[8]:
```
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__',
'__builtin__', '__builtins__', '_ih', '_oh', '_dh', 'In', 'Out', 'get_ipython',
'exit', 'quit', 'open', '_', '__', '___', '__vsc_ipynb_file__', '_i', '_ii',
'_iii', '_i1', '_1', '_i2', '_i3', '_3', '_i4', '_4', '_i5', '_5', '_i6', '_6',
'_i7', '_7', '_i8'])
```

[9]:
```
locals().keys()
```

[9]:
```
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__',
'__builtin__', '__builtins__', '_ih', '_oh', '_dh', 'In', 'Out', 'get_ipython',
'exit', 'quit', 'open', '_', '__', '___', '__vsc_ipynb_file__', '_i', '_ii',
'_iii', '_i1', '_1', '_i2', '_i3', '_3', '_i4', '_4', '_i5', '_5', '_i6', '_6',
'_i7', '_7', '_i8', '_8', '_i9'])
```

Notice how these two outputs are the same. In this notebook, when we're not in a function the local scope is the same as the global scope, however if we access these in a function, we should expect to see something different.

[10]:
```
def first(x):
    def second(y):
        # Let's describe LEGB relative to this point in the code
        return x + y

    return second(4)
```

Relative to that comment **L (local):** `y` is local because it exists within second. **E (enclosed):** `x` is enclosed relative to the comment, because it is local to the scope (first) that *encloses* the scope of second. **G (global):** something like `__name__` is still global here. **B (builtins):** anyting that is a builtin type to python is still built in here like `int`. See notes for an example that verifies this with calls to `locals()` and `globals()` at various points.

1.0.4 Modules and Importing

```
[11]: 'math' in dir()
```

```
[11]: False
```

```
[13]: import math
      'math' in dir()
```

```
[13]: True
```

In this case, math is a `module` which when imported because accessible for us to use, thus we can do things like `math.sqrt(9)`. On the other hand we can see what happens when we use `from` to import.

```
[15]: 'sqrt' in dir()
```

```
[15]: False
```

```
[16]: from math import sqrt
      'sqrt' in dir()
```

```
[16]: True
```

In summary, when we import a module, the *module* becomes accessible, however when we import something from a module, that thing we imported becomes accessible. **Note:** It is also possible to import modules into a scope other than the global one. Although we usually import at the top of a file, if it's done in the scope of a function, whatever is imported is only available within the scope of said function.

Practice Q: What is the difference between `import module` and `from module import *`, and why is the latter statement often problematic?

When we `import module`, `module` becomes accessible for us to use its members, however `from module import *` takes everything inside module and dumps it into our accessible directory. This could be problematic `module` had a method called `foo`, but our own module also had a similarly named method. Instead of importing everything, `import module` allows us to do `module.foo` instead.

02-classes-objects

March 21, 2024

1 Classes and Objects

1.0.1 Attributes

Objects in Python have *attributes*. Even objects like modules have attributes, which usually come in the form of functions or classes (since that's usually what we want to import), but they can really be anything just like any other object.

[2]:
```
x = 'hello'
x.doesnt_exist()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[2], line 2
      1 x = 'hello'
----> 2 x.doesnt_exist()

AttributeError: 'str' object has no attribute 'doesnt_exist'
```

[3]:
```
a = x.also_doesnt_exist
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[3], line 1
----> 1 a = x.also_doesnt_exist

AttributeError: 'str' object has no attribute 'also_doesnt_exist'
```

When we try to access an attribute of an object that doesn't exist, whether it be a function or some other value, we get an `AttributeError`

If we want to see all the attributes of an object we can use `object.__dict__` to access its attributes as a dictionary.

[5]:
```
class Thing:
    def __init__(self, x):
        self.x = x
    def display(self):
        print(self.x)

something = Thing(1234)
something.y = 'hi'
something.__dict__
```

[5]:
```
{'x': 1234, 'y': 'hi'}
```

Notice however that the methods `__init__` and `display` are not in this dictionary. This is because they belong to the class `Thing` as opposed to each instance of thing, because the call `something.display()` is synonymous with `Thing.display(something)` (where something is bound to the self parameter)

This does mean that we should see these methods in the `__dict__` of the class itself, because types are also objects.

[7]:
```
Thing.__dict__
```

[7]:
```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.Thing.__init__(self, x)>,
              'display': <function __main__.Thing.display(self)>,
              '__dict__': <attribute '__dict__' of 'Thing' objects>,
              '__weakref__': <attribute '__weakref__' of 'Thing' objects>,
              '__doc__': None})
```

Note that this is a mappingproxy and not a dict. We can access it similar to a dictionary but we can't write to it like a dictionary. Note these other dunders:

- `__module__` - the module that Thing was defined
- `__doc__` - the docstring of Thing
- `__annotations__` - annotations on Thing's attributes

1.0.2 Accessing attributes of objects and classes

1. When a value is defined in a class, regardless of if it's a `def` or assignment, it is a *class attribute*
2. If you store any value in an object, it's an *object attribute*
3. If you access the attribute of an object, it checks the object first, then its class.
4. If you access the attribute of a class, it just checks if the class has those attributes

If an attribute that we're looking for doesn't exist in the context that we were searching for, we get an `AttributeError`

1.0.3 Static methods and class methods

Similar to how classes can have attributes that store values that apply to the class as a whole, we can also have static methods that behave in a similar vein.

[3]:
```python
class SomethingElse:
    class_attribute = 0

    def __init__(self, value):
        SomethingElse.class_attribute += value

    @staticmethod
    def get_value():
        return SomethingElse.class_attribute
```

The `@staticmethod` decorator in this case is applied to `get_value`. All of these methods belong to the class SomethingElse, however the static method lacks a self parameter, so when it is called from an instance of SomethingElse, the instance isn't *bound* to it.

[5]:
```python
a = SomethingElse(2)
b = SomethingElse(3)
print(a.get_value(), b.get_value(), SomethingElse.get_value())
# all 3 calls here are functionally indistinguishable, following the rules for␣
 ↪class attributes detailed above
```

```
5 5 5
```

Class methods are similar to static methods, however they apply to the class as a whole rather than an instance.

[6]:
```python
class AnotherThing:
    def method1(self):
        pass # Instances are bound to self

    @staticmethod
    def method2():
        pass # Nothing is bound

    @classmethod
    def method3(cls):
        pass # Class is bound
```

A use case for class methods are *factory methods* which creates an object of a type. For example, I may have a class called `Vector` which given certain values and the class can return a `Vector` object to me. *mathematical vector not c++

03-functions-parameters

March 21, 2024

1 Functions and Their Parameters

For clarification, *parameters* are part of the method signature or the variables listed in the function definition. On the other hand *arguments* are the actual values passed in to the function.

Parameter flexibility: A good example of a function with flexible inputs is `print`, for e.g. `print('hello world')`, `print('hello', 'world')`, `print('hello', 'world', end='!')` are all valid print statements.

The two types of arguments 1. *Positional Arguments*, which get matched to their corresponding parameters based on their *position* or the order by which they are passed in. 2. *Keyword Arguments*, which are matched to their corresponding parameters based on how they map to the name of an existing parameter.

1.0.1 Positional Arguments

Suppose we had the following function:

[2]:
```
def y(m, x, b):
    return m * x + b
```

We can pass arguments positionally, in the order of `m, x, b` with comma separation, or we can unpack values with `*` to be passed into those positions.

[7]:
```
# The unpacking must be done on something iterable
some_tuple = (3, 4, 5)
some_list = [3, 4, 5]

print(y(3, 4, 5), y(*some_tuple), y(*some_list), y(*range(3, 6)))
```

```
17 17 17 17
```

[8]:
```
# We can also unpack only some of the arguments
other_tuple = (4, 5)
y(3, *other_tuple)
```

[8]:
```
17
```

[9]:
```
# And as expected, unpacking the wrong number of arguments is bad
y(*range(3, 7)) # the same as trying to do y(3, 4, 5, 6)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 2
      1 # And as expected, unpacking the wrong number of arguments is bad
----> 2 y(*range(3, 7)) # the same as trying to do y(3, 4, 5, 6)

TypeError: y() takes 3 positional arguments but 4 were given
```

1.0.2 Keyword Arguments

But we can also pass in arguments with keywords as such:

[12]:
```
y(x = 4, b = 5, m = 3) # positionally these are in the order m, x, b
```

[12]:
```
17
```

This means we can also unpack dictionaries to become keyword arguments:

[16]:
```
some_dictionary = {'x': 4, 'b': 5, 'm': 3}
y(**some_dictionary)
```

[16]:
```
17
```

Notably we need to use `**` to unpack this dictionary. If we had done `*` instead, it would have unpacked the keys of the dictionary, which would have been the equivalent of `y('x', 'b', 'm')` (obviously not what we want). This makes sense, because dictionaries are also iterable just like lists and tuples, however iterating over them generally iterates over its keys.

1.0.3 Positional and Keyword arguments in unison

We can also use both of these types of arguments at the same time, following certain rules: Most importantly, positional arguments must all come before keyword arguments (otherwise how would we know what order they are in).

[18]:
```
y(3, **{'b': 5, 'x': 4}) # this is fine
```

[18]:
```
17
```

[20]:
```
y(**{'b': 5, 'x': 4}, 3) # this is not, despite it being clear what I'm trying␣
↪to do
```

```
  Cell In[20], line 1
    y(**{'b': 5, 'x': 4}, 3) # this is not, despite it being clear what I'm␣
↪trying to do
                          ^
```

```
SyntaxError: positional argument follows keyword argument unpacking
```

1.0.4 Designing the parameters

Since there are different ways we can pass in arguments, there are also different ways we can design our function's parameters to accept said arguments.

1. Default arguments We can give parameters default arguments, with the syntax below. A few things to note: - If a parameter is given a default argument, then all subsequent parameters must have them as well. This makes sense, because if they have default parameters, they're essentially optional and once we specify one, the idea of positionality is sort of lost. - It's generally bad practice to use mutable types as default arguments. When we define a function with default arguments they are stored under `func.__defaults__`, which can carry over between calls in erroneous ways.

[2]:
```
def add(first, second = None):
    if second is not None:
        return first + second

    return first

print(add(1, 2), add(5))
```

```
3 5
```

2. Variable number of arguments This is essentially the opposite of unpacking. Instead of `*(a, b, ..., c)` translating to `a, b, ..., c`, we can do it the other way around. A parameter that allows this is called a *tuple-packing parameter*:

[5]:
```
def some_func(x, *args):
    print(type(args), len(args), args)

some_func('hello', 'world', 'foo', 'bar', 1234, True)
```

```
<class 'tuple'> 5 ('world', 'foo', 'bar', 1234, True)
```

Note that 'hello' is not in the tuple because it was passed into x. Then all *remaining* arguments get packed into the parameter `args`.

3. Setting positional and keyword requirements The example above then begs the question, what happens if we define parameters after the tuple-packing parameter. Parameters that follow can only be passed by *keyword*, because as we would imagine, if all remaining positional arguments are packed into `args`, then the only way to specify everything else must be keyword arguments.

This idea leads to the following syntax, which means everything after the `*` must be passed as a keyword argument.

[6]:
```
def valid_func(a, b, *, c):
    pass
```

In this case a and b can be passed however we want. I think it's easiest to think of the * as an arbitrary tuple-packing parameter that we submit all remaining positional arguments into, such that c must be passed by keyword, although it's important to note that this isn't entirely true, just a potentially useful way to think about it:

[8]:
```
valid_func(1, 1, 2, c = 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 valid_func(1, 1, 2, c = 3)

TypeError: valid_func() takes 2 positional arguments but 3 positional arguments
 ↪(and 1 keyword-only argument) were given
```

[11]:
```
def not_valid_func(*args, *, some_kwarg):
    pass
```

```
  Cell In[11], line 1
    def not_valid_func(*args, *, some_kwarg):
                              ^
SyntaxError: * argument may appear only once
```

In the example above, a and b could be passed either positionally or as keywords, so there's another rule we can establish: Listing / as a parameter makes everything on its left positional only.

[ ]:
```
def still_valid(a, /, b, *, c):
    pass
```

In `still_valid`, `a` must be passed positionally, `b` can be passed however, and `c` must be a keyword argument.

4. Dictionary packing Since we could pack positional arguments into tuples, there's no reason we shouldn't be able to pack keyword arguments into dictionaries:

[9]:
```
def func(*args, **kwargs):
    print(args)
    print(kwargs)

func('hello', 12, number = 42, truth = False, name = 'boo')
```

```
('hello', 12)
{'number': 42, 'truth': False, 'name': 'boo'}
```

Final self-consistent observations: - Only one tuple-packing parameter can be listed, and no positional parameters can follow it. - Only one dictionary-packing parameter can be listed, and no parameters at all can follow it.

04-context-managers

March 21, 2024

1 Context Managers

Directly from the notes because it's important: A very common software requirement is one you might call automatic wrap-up, which is to say that sometimes our programs perform operations where certain things need to be finalized or unwound when the operations have finished, whether the operations themselves succeeded or failed.

Suppose we wanted to read from a file. We can use a `try` to try and open the file, and `finally` to close it if it ever opened successfully. However, for more complicated problems this process can get tricky so we can instead use *context managers* using a `with` statement.

```
with open(some_file_path, 'r', encoding='utf-8') as some_file:
    do_stuff(some_file)
```

Consider this example. The open function returns a file object stored into `some_file`, and this entire operation is encapsulated within the `with` statement.

As it turns out, file objects are *context managers* so it has predefined some operations to perform when we exit this context, notably whether or not we should close the file based on whether opening it succeeded in the first place.

A common example is when we are unit testing and expect some code to fail in a particular way:

```
with self.assertRaises(SomeError):
    thing_that_triggers_some_error()
```

Unlike the file opening example we dont use `as` because we don't actually care about doing anything with the context manager in the body.

1.0.1 The `contextlib` module

This standard library module basically contains some context managers that might be useful to us, for example capturing standard input (particularly useful for testing):

[6]:
```
import contextlib
import io
with contextlib.redirect_stdout(io.StringIO()) as output:
    print('hey there')
    # this print is captured by the context manager so doesn't print anything
```

[7]:
```
# unless we ask for it
output.getvalue()
```

```
[7]: 'hey there\n'
```

1.0.2 Building context managers

Any object can be a context manager *if and only if* it satisfies certain properties. We can call these properties the context manager protocol, and anything that supports these operations can function as as context manager: 1. `__enter__(self)` - is called on the object as the `with` statement is entered. The return value of this function is stored into a variable if we use `as something`. 2. `__exit__(self, exc_type, exc_value, exc_traceback)` - is the opposite of enter. When the context manager exits, it passes in certain values to these parameters. If the `with` statement exits with no issues, all of these will get passed `None` as an argument, otherwise they will contain the type, value, and traceback of the exception respectively.

```
[11]: class SomeContextManager:
          def __init__(self, value):
              self.value = value
              print('constructed')

          def __enter__(self):
              print('entering')
              return self

          def __exit__(self, exc_type, exc_value, exc_traceback):
              if exc_type == None:
                  print('successful exit')
              else:
                  print(f'unsuccessful exit: {exc_type}')
                  return True
```

```
[12]: with SomeContextManager('hello world') as x:
          print(x.value)
```

```
constructed
entering
hello world
successful exit
```

```
[13]: with SomeContextManager('goodbye world') as x:
          raise Exception
```

```
constructed
entering
unsuccessful exit: <class 'Exception'>
```

The reason an error does not come back in the second example, is because `__exit__` returned `True`, which basically tells the context manager "I have handled the issue so suppress it"

05-asymptotic-analysis

March 21, 2024

1 Asymptotic Analysis

1.0.1 Why

Part of writing code, is we want to write code that is efficient. Two important metrics we care about are space (memory usage), and time. We could analyze every single line of code down to it's very last detail, accounting for how fast our processor may be and try to precisely estimate how much time something will take, however we won't always have access to all this information (such as what computer our code will run on, or what our input sizes look like). Thus, we use asymptotic analysis to measure the *complexity* of code, which allows us to compare different algorithms assuming our input size can get really large.

1.0.2 Big O notation

By definition, $f(n)$ is $O(g(n))$ *if and only if* there are positive constants c and n_0 such that $f(n) \leq cg(n) \forall n \geq n_0$. Which is just simply saying, ignoring constant factors, as n gets really big, $f(n)$ will always be less than $g(n)$. This makes sense, because when we can't identify small things that affect constant factor like how long it takes to perform an addition operation, we can more about the big picture, as in how do the growth rate of different functions compare as n gets really large.

Example: Show that the function $f(n) = 3n + 4$ is $O(n)$

Proof.

$$\begin{aligned}
f(n) &\leq cg(n) && \text{for all } n \geq n_0 && \text{(By Big-O definition)} && (1)\\
3n + 4 &\leq cn && \text{for all } n \geq n_0 && && (2)\\
&\text{Let } c = 4, n_0 = 4 && \text{(Select constants)} && && (3)\\
3n + 4 &\leq 4n && \text{for all } n \geq 4 && && (4)\\
4 &\leq n && \text{for all } n \geq 4 && \blacksquare && (5)
\end{aligned}$$

It's worth noting that a function $f(n) = 3n + 4$ is $O(n)$, but also $O(n^2)$, $O(n^n)$, and $O(n \log n)$, it's just that $O(n)$ is what we consider the "closest-fit".

We also don't write the bases of logarithms in Big-O, because the difference between bases is constant:

$$\frac{\log_a n}{\log_b n} = \log_a b$$

1.0.3 Examples in Code/Practice Analysis

[10]:
```
def add_nums(nums):
    for num in nums:
        for x in range(5):
            print(num + x, end=' ')

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

```
1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 5 6 7 8 9
```

`add_nums`

Time: $O(n)$ Although there are two for loops, the inner loop runs for a constantly defined amount of iterations, and $5n$ has a closest-fit of $O(5n)$.

Space: $O(1)$ *num* and x both require constant amounts of space, and even though we are working with a list, we don't use any auxiliary memory within the function since the space for the list was allocated outside of the scope of `add_nums` itself.

[7]:
```
def multiplication_table(n):
    for i in range(1, n + 1):
        print(*[i * j for j in range(1, n + 1)])

multiplication_table(5)
```

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

`multiplication_table`

Time: $O(n^2)$

Although it seems as thought there are n prints occuring due to the for loop, there is a second for loop within the list comprehension that also runs for n time, so this is actually $n \times n$, or $O(n^2)$.

Space: $O(n)$

Even though this function could have been implemented with $O(1)$ space, it still uses $O(n)$, because each call of the list comprehension allocates $O(n)$ memory to generate the list, before it is unpacked and printed. Since we don't store the comprehension anywhere, for each pass of the loop it is created and then discarded after printing.

It's worth noting that this function could have been implemented by simply printing directly instead of using a comprehension, which would still have been $O(n^2)$ time, but $O(1)$ space instead.

[5]:
```
def sieve(num):
    prime = [True for i in range(num+1)]
    p = 2
    while (p * p <= num):
        if (prime[p] == True):
            for i in range(p * p, num+1, p):
                prime[i] = False
        p += 1

    for p in range(2, num+1):
        if prime[p]:
            print(p, end=' ')

sieve(100)
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

`sieve`

Time: $O(n \log \log n)$

Constructing `prime` with a comprehension is $O(n)$. Although the outer while loop seems to run in $\sqrt{n}$ time, we need to consider the inner for loop. This loop marks multiples of p as not prime from p^2 to n, so for each prime we encounter the operation is $O(\frac{n}{p})$. The notable thing here is we don't actually process all $\sqrt{n}$ numbers, since each number is marked exacctly once by its smallest prime factor.

We can approximate the work done for each prime p by this harmonic series to compute the total complexity:

$$O\left(n\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+\frac{1}{11}+...\right)\right)$$

This series based on how primes are distributed is approximately $O(n \log \log n)$

Then finally going through n numbers and checking if they are prime and printing if it is is just $O(n)$.

Space: $O(n)$

The only auxiliary space we use in this code is the list `prime` which uses a linear amount of space relative to the input of `num`

06-searching

March 21, 2024

1 Searching

Searching is simply the process of finding some data that we've stored in some sort of data structure.

Suppose we have a list of integers in Python, in which we want to find where some element exists, if it exists at all. Assuming we know nothing else about this list, we can apply sequential search, or literally traversing the list until we find what we want, (or don't).

[ ]:
```python
def sequential_search(items, key):
    for i in range(len(items)):
        if items[i] == key:
            return i

    return None
```

First of all this method uses $O(1)$ memory, as we only allocate space for things like `i` which is purely constant.

Time wise however, this search is performed in $O(n)$, because we search through each sequential element once. In the worst case where we don't find our key, we have to look through a total of n things.

Can we do better? No. Or at least not without other information, since we don't know anything about the contents of the array. But why? Let's prove this by contradiction.

Suppose I could improve this algorithm, such that it performed in better of $O(n)$ time. Let's say in this algorithm I compared `key` to k different items. Since this algorithm is strictly better than $O(n)$, then $k < n$. If $k < n$, there must exist at least 1 element I didn't compare with. What if that one element I didn't compare with was my key, and the other k elements were not? Then my search would have terminated incorrectly, therefore this improvement cannot exist.

1.0.1 Binary Search

So instead, let's improve this algorithm anyways. What if we knew our list was sorted?

We can then divide the entire list in half, and compare our key with the middle. If it is exactly there, great, but if it's not since the list is sorted we can identify which half of the list it's in. Then within that half, we can repeat the process until we find it.

[1]:
```python
def binary_search(items, key):
    l = 0
    r = len(items) - 1

    while l <= r:
        mid = (l + r) // 2

        if items[mid] == key:
            return mid
        elif items[mid] > key:
            r = mid - 1
        else:
            l = mid + 1

    return None
```

This method still only requires $O(1)$ memory, because we only allocate space for variables like `l`, `r`, or `mid`.

However, this function has a time complexity of $O(\log n)$, which is considerably better than $O(n)$ for large values of n. For every iteration of the loop where we don't find the key, we can essentially eliminate half of the list. Thus, this loop will iterate in logarithmic time.

But suppose I wanted to implement this recursively, because I see the recursive structure in reperforming a smaller binary search on the side I divide to.

[2]:
```
def binary_search(items, key, l, r):
    if l > r:
        return None

    mid = (l + r) // 2

    if items[mid] == key:
        return mid
    elif items[mid] > key:
        return binary_search(items, key, l, mid - 1)
    else:
        return binary_search(items, key, mid + 1, r)
```

Unfortunately, although this has the same runtime, the space complexity of this search has now increased from $O(1)$ to $O(\log n)$. Each call to a function is stored as a stack frame on our call stack, and by replacing iterations of a while loop with recursive calls, we've essentially populated the call stack with as many frames as there would have been iterations of the while loop.

07-databases

March 21, 2024

1 Databases

When we want to store data outside of our program, we might use a database to store this information. In this case we're using SQL (structured query language), in this case a database management system called SQLite that utilizes SQL to manipulate databases.

1.0.1 Relational databases

A relational database allows us to store items that can be related to others in some way.

This data is usually structured in the form of tables, think excel spreadsheet with rows and columns.

courses

row	course_number	course_name	unit_count
1	ICS 31	Introduction to Programming	4
2	ICS 32	Programming with Software Libraries	4
3	ICS 33	Intermediate Programming with Python	4

Each row of the table is some entry of data, and can essentially correspond to a tuple in Python, where the table defines the types stored in the tuple. It's almost as if the table above corresponds to the following namedtuple (although in this example I used NamedTuple from typing as opposed to namedtuple from collections):

[1]:
```python
from typing import NamedTuple

class Course(NamedTuple):
    row: int
    course_number: str
    course_name: str
    unit_count: int

# e.g.
ics_31 = Course(row = 1, course_number = 'ICS 31', course_name = 'Introduction␣
 ↪to Programming', unit_count = 4)
ics_31
```

[1]:
```
Course(row=1, course_number='ICS 31', course_name='Introduction to Programming',
unit_count=4)
```

1.0.2 Primary Keys

A *primary key* is a subset of columns that are guaranteed to be unique for different rows. Consider storing a bunch of people in a database. We can't really use their names, because with a lot of people there's almost guaranteed to be two people who share identical names. This is why many places assign people IDs, like a student ID, or taxpayer ID to *uniquely* identify you.

When we use a DBMS like sqlite, we can specify which columns represent the primary key and it will guarantee that: 1. No two rows have identical primary keys 2. We can associate a primary key to a row to find an entry quickly

The mechanism that a DBMS utilizes to guarantee this is quite similar to dictionaries, where are primary key would be the primary key of a row.

1.0.3 Relationships

Since we can uniquely and quickly identify rows in a table with primary keys, it makes sense that we should be able to relate different elements to another through these primary keys.

Relationships have a *cardinality*, which is a limit on how many elements can have relationships with each other. Suppose we have tables A, B (can sort of think of them as mathematical sets).

1. **one-to-one** - an element of A can only be related to at most one element in B, and vise versa. e.g. (ignoring dual citizenship) one person can be associated to one passport, and vise versa
2. **one-to-many** - an element of A can be related to many elements of B, but each element of B can only be related to one element in A. e.g. a library member can have many books checked out a once, but any checked out book can only have been checked out by a single library member at once.
3. **many-to-many** - an element of A can be related to many elements in B, and vise versa. e.g. I can "favorite" many songs on Spotify, and any song on Spotify can be favorited by many people. (although probably not the best example in terms of databases)

Many-to-many is the *most* flexible, but when considering desgining relationships, we only want to be a flexible as we have to be.

1.0.4 Foreign Keys

In order to actual implement these relationships, it makes sense to store what's called a *foreign key*, which is essentially the primary key of something you want to reference.

Usually with a DBMS we explicitly specify a column to store foreign keys, so it can enforce "referential integrity" which means what each row refers to should actually exist.

Since columns store scalar values, many-to-many relationships usually need to be implemented with some intermediary table that contains the foreign keys of related rows.

1.0.5 SQL and SQLite

How do we apply this to actually managing a SQL db in python.

[2]:
```python
import sqlite3
connection = sqlite3.connect(':memory:')
type(connection)
```

[2]:
```
sqlite3.Connection
```

This syntax allows us to establish a 'connection' with a database. In this case `':memory:'` denotes an in-memory database, but we could have instead passed in a filesystem path. This connection is our portal to execute SQL queries against our databases.

[3]:
```python
connection.execute(
    '''
    CREATE TABLE person(
        person_id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
    ) STRICT;
    '''
)
```

[3]:
```
<sqlite3.Cursor at 0x1079bdac0>
```

Calling this `execute` method is how we pass SQL statements as a string for the sqlite to interpret and operate on our database. A few things to note about this specific query: - **SQL keywords** or the tokens in caps are keywords that have some meaning in the SQL language - Whitespaces and indentation don't matter which is why this query is valid despite being a weirdly formatted string - Each statement is concluded with a semi colon

This syntax **creates** a **table** called **person**, which has the listed columns, with the types specified next to it, as well as `PRIMARY KEY` indicating that we want `person_id` to be a primary key.

Also the `STRICT` keyword means we will strictly enforce the types stored in these columns.

Also also, notice that this execute returned a `<sqlite3.Cursor>` object. If our statement had queried for data, it would be stored in said cursor, but creating a table doesn't so we don't really need to do anything with it.

[4]:
```python
cursor = connection.execute('SELECT name FROM sqlite_schema;')
print(cursor.fetchone())
print(cursor.fetchone())
cursor.close()
```

```
('person',)
None
```

SQLite retains certain metadata about how our db is structured, which we can also query for from the `sqlite_schema` table. What we've essentially done is `SELECT`ed all the rows, only asking for the `name` from each row, `FROM` the table `sqlite_schema`. Which only returned `('person',)`, since that's the only table we have.

The method `cursor.fetchone()` allows us to get these results one at a time, and it returns `NONE` once there are none left. Then we make sure to close the cursor when done.

1.0.6 Insertion

Now that we've created a table, we presumably want to insert data to populated it with values.

[5]:
```
connection.execute(
    '''
    INSERT INTO person (person_id, name, age)
    VALUES (1, 'Boo', 13);
    '''
)
```

[5]:
```
<sqlite3.Cursor at 0x107a01d40>
```

`INSERT INTO x` indicates we want to insert into table `x`, and we intend to specify the columns in the following tuple `(person_id, name, age)` Then the keyword `VALUES` indicates the corresponding values we want to insert into those columns.

[6]:
```
connection.execute(
    '''
    INSERT INTO person (person_id, name, age)
    VALUES (1, 'Alex', 47);
    '''
)
```

```
---------------------------------------------------------------------------
IntegrityError                            Traceback (most recent call last)
Cell In[6], line 1
----> 1 connection.execute(
      2     '''
      3     INSERT INTO person (person_id, name, age)
      4     VALUES (1, 'Alex', 47);
      5     '''
      6 )

IntegrityError: UNIQUE constraint failed: person.person_id
```

Notice how the above error fails with message "UNIQUE constraint failed", because we had already inserted a row with a `person_id` of 1. This is SQLite at work ensuring the integrity of our database.

[10]:
```
connection.execute(
    '''
    INSERT INTO person (person_id, name, age)
    VALUES (2, 'Alex', 47);
    '''
)
```

[10]: <sqlite3.Cursor at 0x10794a840>

1.0.7 Filtering

When we use SELECT to pull rows from a table, it is quite likely that we don't want to return every row because often we are looking for something specific:

[11]:
```
cursor = connection.execute(
    '''
    SELECT name FROM person
    WHERE age < 40;
    '''
)
print(*cursor.fetchall())
cursor.close()
```

```
('Boo',)
```

Generally the same syntax as before, however we used the WHERE keyword to specify a boolean expression that filtered out results we didn't want.

[12]:
```
cursor = connection.execute(
    '''
    SELECT name, age FROM person
    WHERE length(name) = 4 AND person_id BETWEEN 1 AND 10;
    '''
)
print(*cursor.fetchall())
cursor.close()
```

```
('Alex', 47)
```

We can also use truthiness as a boolean, where non-zero integers are true, although it is probably best to avoid this:

[13]:
```
cursor = connection.execute(
    '''
    SELECT name, age FROM person
    WHERE age;
    '''
)
print(*cursor.fetchall())
cursor.close()
```

```
('Boo', 13) ('Alex', 47)
```

1.0.8 Ordering

On top of filtering, we can also control the order that the SELECT returns data with ORDER BY. Note, if we do both, the WHERE must come prior to the SELECT:

[14]:
```
cursor = connection.execute(
    '''
    SELECT name, age FROM person
    ORDER BY age DESC;
    ''' # DESC for descending, ASC for ascending
)
print(*cursor.fetchall())
cursor.close()
```

```
('Alex', 47) ('Boo', 13)
```

1.0.9 Modifying data

When we're managing a database, it also stands to reason that we'd want to modify existing rows. This can be done with `UPDATE` that requires us to specify `SET`: which is what we want to set, and `WHERE`: which is where we want to set these changes

[15]:
```
connection.execute(
    '''
    UPDATE person
    SET age = age + 1
    WHERE person_id = 2
    '''
)

cursor = connection.execute(
    '''
    SELECT name, age FROM person
    '''
)
print(*cursor.fetchall())
cursor.close()
```

```
('Boo', 13) ('Alex', 48)
```

We can also update multiple values as such:

[16]:
```
connection.execute(
    '''
    UPDATE person
    SET age = 25, name = 'Bob'
    WHERE person_id = 2;
    '''
)

cursor = connection.execute(
    '''
    SELECT name, age FROM person
    '''
)
print(*cursor.fetchall())
cursor.close()
```

```
('Boo', 13) ('Bob', 25)
```

If we don't specify a `WHERE`, it will simply apply `SET` to every row in the table.

1.0.10 Deleting things

We can delete rows from tables, as well as tables entirely

[17]:
```
connection.execute(
    '''
    DELETE FROM person
    WHERE person_id = 2;
    '''
)
```

[17]:
```
<sqlite3.Cursor at 0x107949ec0>
```

Or to drop an entire table

```
DROP TABLE person;
```

From this point on (just for databases) I'll write examples in markdown instead of executing actual Python just for notes, but these are worth experimenting with.

1.0.11 NULL

NULL is a value that can be stored that represents the lack of a value. When we fetch data, it translates back to Python as `None` since that seems to make the most sense.

Suppose we have colums a, b, c, all integers and we add the following entry:

```
INSERT into some_table (a, b)
VALUES (1, 2);
```

Fetching this entry form the table would yield the tuple `(1, 2, None)`, since a NULL was stored where we didn't specify a value in c.

We can also use NULL with WHERE to find empty rows in columns:

```
WHERE some_column IS NULL;
```

Notice how we can't do `some_column = NULL`.

When we construct a table, we can also use the keyword `NOT NULL` (similar to `PRIMARY KEY`) to specify that column cannot contain NULL values.

1.0.12 Relationships

How do we actually apply theoretically building relationships into SQL queries:

```
CREATE TABLE othertable(
    other_id INTEGER NOT NULL PRIMARY KEY
) STRICT;

CREATE TABLE mytable(
    id INTEGER NOT NULL PRIMARY KEY,
    name TEXT,
    related_thing INTEGER NOT NULL,
    FOREIGN KEY (related_thing) REFERENCES othertable(other_id)
) STRICT;
```

Using this syntax we can specify a certain column as a foreign key, which relates to some other row. We can see that this is sort of a one way relationship, so we can build one-to-one and one-to-many quite easily, however for many-to-many we'd likely have to create an intermediary table. (the example in the notes is better than anything I can come up with)

1.0.13 Joins

Once we establish foreign key relationships, we can use joins to combine the rows of tables. The most common type is the "inner join". Once again, I would say reference the notes because it's explained much thoroughly in a way that I don't think can be summarized adequately.

08-comprehensions

March 21, 2024

1 Comprehensions

Comprehensions are a feature of python that simplify the syntax for populating certain data structures with values.

1.0.1 List Comprehensions

Suppose I wanted to construct a list representing the squares of numbers 1 through 10, I might do the following:

```
[1]: square = []

     for i in range(1, 11):
         square.append(i * i)

     square
```

```
[1]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

This sort of process of looping through a list, and appending some value that is a function of what we're iterating is quite common, so instead we can do the same thing with a list comprehension:

```
[2]: also_square = [i * i for i in range(1, 11)]
     square == also_square
```

```
[2]: True
```

And we can also apply conditions to the comprehension: suppose we wanted to produce the same squares but only for even numbers:

```
[3]: [i * i for i in range(1, 11) if i % 2 == 0]
```

```
[3]: [4, 16, 36, 64, 100]
```

```
[ ]:
```

This syntax is in fact quite powerful since we can do this with any iterable, so we can do some interesting fancy things:

```
[4]: [x * y for x in 'ABC' for y in (1, 2, 3)]
```

```
[4]: ['A', 'AA', 'AAA', 'B', 'BB', 'BBB', 'C', 'CC', 'CCC']
```

This same syntax can be applied for sets, as well as tuples, although notably for tuples we must enclose the comprehension with `tupe(some_comprehension)` as opposed to `()`, which would produce a generator.

And a similar thing can be done with dictionaries:

```
[5]: {s: len(s) for s in ['hello', 'world', 'a', '', '123']}
```

```
[5]: {'hello': 5, 'world': 5, 'a': 1, '': 0, '123': 3}
```

09-iteration

March 21, 2024

1 Iteration

Iteration is generally goverened by two types of objects. *Iterables* and *Iterators*. *Iterables* is a type of object that follows a certain protocol which allows it to be iterated over, for example using a for loop to traverse its contents. An *iterator* is a type of object that manages the process of producing certain values one at a time.

```
[1]: for x in [1, 2, 3, 'a', 'b', 'c']:
         print(x, end=' ') # lists are iterable
```

```
1 2 3 a b c
```

We can look at the underlying mechanism by which loops might perform iteration. We can create an iterator from our iterable, and ask it to produce sequential values by calling `next`

```
[2]: some_numbers = [1, 2, 3]
     it = iter(some_numbers)
```

```
[3]: next(it)
```

```
[3]: 1
```

```
[4]: next(it)
```

```
[4]: 2
```

```
[5]: next(it)
```

```
[5]: 3
```

```
[6]: next(it)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[6], line 1
----> 1 next(it)

StopIteration:
```

Once an iterator fails to yield further values to us, it raises `StopIteration` and that's how we know it has completed.

1.0.1 Building an Iterable

Now suppose we want to construct an object that is iterable, or implements the *iterable protocol*. There are fundamentally two required methods:

- `__next__(self)` - returns next element from an iterator, or raises `StopIteration`
- `__iter__(self)` - constructs and returns an iterator, to iterate over itself

An **iterable** must simply implement the latter that returns an iterator. An **iterator** implements *both* methods. Even though the iterator itself is already assumed to implement next, if we try to use something like a for loop, python attempts to call `iter()` on that object.

(Alternatively implementing `__len__` and `__getitem__` also makes an object iterable)

We can observe how lists generally follow this behavior:

[13]:
```
x = some_numbers.__iter__()
x.__next__()
```

[13]:
```
1
```

A good example given in the notes is building something similar to the builtin `range()` method in Python. Here is a simplified version.

[1]:
```
class FauxRangeIter:
    def __init__(self, faux_range):
        self._faux_range = faux_range
        self._cur = faux_range.start

    def __iter__(self):
        return self

    def __next__(self):
        if self._cur >= self._faux_range.stop:
            raise StopIteration
        else:
            temp = self._cur
            self._cur += self._faux_range.step
            return temp

class FauxRange:
    def __init__(self, start, stop = None, step = None):
        if stop is None:
            stop = start
            start = 0

        if step is None:
            step = 1

        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        return FauxRangeIter(self)
```

For the most part, this will behave similarly to an actual `range`, although it isn't a generator. When a for loop iterates over a `FauxRange`, it gets an iterator by calling `__iter__`, then sequentially calls next until it hits a StopIteration

[2]:
```
for x in FauxRange(3, 10, 2):
    print(x, end=' ')
```

```
3 5 7 9
```

It's also worth noting that pretty much all of the methods in the two classes above run in $O(1)$ time, because if we lazily produce values as we need them on calls of `__next__`, we can be considerably more efficient than were we to figure out all of the values first and store them somewhere.

10-generators

March 21, 2024

1 Generators

Generators are a tool in Python that allow you to quickly create *iterators* with functions.

```
[3]: def some_generator():
         for x in range(3):
             yield x

     sg = some_generator()
```

```
[4]: sg
```

```
[4]: <generator object some_generator at 0x105323040>
```

Instead of using `return`, generators use the `yield` statement, which is in some ways an intermediary return. Calling the generator function returns a generator object (an iterator).

Every time `__next__` is called on this iterator, it executes the code within the function until a `yield` is reached. It then pauses execution of that function and *yields* said value. Once the function exits, it raises `StopIteration`

```
[5]: next(sg)
```

```
[5]: 0
```

```
[6]: next(sg)
```

```
[6]: 1
```

```
[7]: next(sg)
```

```
[7]: 2
```

```
[8]: next(sg)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[8], line 1
----> 1 next(sg)
```

```
StopIteration:
```

This can be a powerful tool for building iterators, because of our ability to lazily yield values as we need them. The `range` function is an example of a type that behaves like generator, because it doesn't compute everything in a range at once, instead it computes the values as we need them which is why it can be incredibly efficient for large ranges.

Any time a generator function exits, whether it be reaching the end, or manually calling `return`, it raises `StopIteration`, and if we return a value, that value is given to the StopIteration exception.

11-using-generators

March 21, 2024

1 Using Generators

Generators:

1. Isolate how we generate a sequence and what we do with it
2. Lazily produce values
3. Can stop generating for arbitrary reasons

We can also build generators with comprehensions, consider the examples below which are basically equivalent, but different syntactically:

[1]:
```
def squares_generator():
    for x in range(10):
        yield x * x

first_generator = squares_generator()
second_generator = (x * x for x in range(10))

list(first_generator) == list(second_generator)
```

[1]:
```
True
```

But what is the point of doing `list(second_generator)`, when we could have applied the same comprehension to a list.

> Suppose we weren't building a list, but instead iterating over some of the values, but not necessarily all of them. A list comprehension would construct every single value, and we would select the one we need, however a generator would only produce as many values as we need.

1.0.1 Infinite generators

We can also legally write generators that yield an infinite amount of values

[2]:
```
def count():
    start = 0
    while True:
        yield start
        start += 1
```

```
counter = count()
next(counter)
```

```
[2]: 0
```

```
[3]: next(counter)
```

```
[3]: 1
```

Even though we technically would generate an infinite amount of values, as long as I only ask for a finite amount of values, the generator will only do a finite amount of work.

Of course this means this would not be allowed:

```
[ ]: for x in counter(): # don't run this block lol
         pass
```

1.0.2 Combining generators

It's very natural that we may have generators that yield values based on values yielded from other generators. For example, we can write a generator that yields a finite amount of values from our infinite generator above.

```
[6]: def take5(thing):
         for _ in range(5):
             yield next(thing)
```

```
[13]: list(take5(count()))

      # casting to list forces take5 to keep yielding until StopIteration
```

```
[13]: [0, 1, 2, 3, 4]
```

Keep in mind this example does not account for `thing` raising StopIteration.

1.0.3 Forwarding everything

If we want to force a generator to yield all of it's values we can use the syntax `yield from`, which can be particularly useful when we don't want to allocate memory to store values, so that it can immediately be forwarded onwards.

A lot of problems that need to be solved in different software are already built in to Python, and utilize iteration. Some are built in, and some are included in the `itertools` module which is part of Python's standard library.

12-python-data-model

March 21, 2024

1 The Python Data Model

Many of Python's features rely on duck typing and the existence of protocols.

1.0.1 Lengths

An object is *sized* if it can be asked for its length. This protocol can be implemented using the `__len__` dunder.

[1]:
```
len('a string') # sized
```

[1]:
```
8
```

[4]:
```
class LengthyThing:
    def __init__(self, x):
        self._x = x

    def __len__(self):
        return self._x

example_thing = LengthyThing(12)
len(example_thing)
```

[4]:
```
12
```

1.0.2 Truthiness

`None` is falsy, non-zero numbers are truthy, strings are truthy when non-empty etc. How does this extrapolate to other objects:

1. `__bool__` - when this dunder is implemented, determines the truthiness of an object, otherwise:
2. `__len__` - when an object is sized, if it has a non zero size it is truthy, otherwise
3. All objects otherwise are considered truthy (the assumption that the existence of an object means some expression's value is not None)

[5]:
```
bool(example_thing)
```

```
[5]: True
```

```
[6]: bool(LengthyThing(0))
```

```
[6]: False
```

1.0.3 Indexing

1.0.4 Indexing

Indexable objects can implement `__getitem__(self, index)`, `__setitem__(self, index)`, and `__delitem__(self, index)`.

```
[10]: class Indexable:
          def __init__(self):
              pass

          def __getitem__(self, index):
              print(f'__getitem__ : {index}')

          def __setitem__(self, index, value):
              print(f'__setitem__ : {index} {value}')

          def __delitem__(self, index):
              print(f'__delitem__ : {index}')
```

```
[11]: indexable_obj = Indexable()

      indexable_obj[2]
      indexable_obj[82] = 'test'
      del indexable_obj[11]
```

```
__getitem__ : 2
__setitem__ : 82 test
__delitem__ : 11
```

1.0.5 Sequences

When an object implements both `__len__` and `__getitem__`, it follows the sequence protocol which allows an object to become iterable without explicitly having an `__iter__` method.

Note from testing: The implicit forward iterator only stops when `__getitem__` raises `IndexError`, and reversed uses `__len__` to determine when to stop. So a forward iterator only stops on IndexError regardless of whether or not `__len__` is implemented.

```
[21]: class SequenceThing:
          def __init__(self, x):
              self._len = x
```

```
    def __getitem__(self, index):
        return index;

    def __len__(self):
        return self._len

list(iter(SequenceThing(3)))
```

[21]: 3

[23]:
```
list(reversed(even_nums))
```

[23]: [4, 2, 0]

When building sequences, we may often want to provide:

1. `__iter__(self)` - to construct custom iteration
2. `__reversed__(self)` - to construct custom reverse iteration
3. `__contains__(self)` - so we can use the `in` keyword on this object

1.0.6 Slicing

Slicing allows us to 'slice' a section of a sequence out of an object. The syntax is similar to indexing however instead of `obj[index]`, it's `obj[start:stop:step]`.

For the indexing methods like `__getitem__`, the value passed into `index` is a slice object, which has attributes start, stop step.

[24]:
```
class Sliceable:
    def __getitem__(self, index):
        print(index)

some_object = Sliceable()
some_object[1]
some_object[1:2:3]
some_object[1::3]
```

```
1
slice(1, 2, 3)
slice(1, None, 3)
```

We can also unpack the three values as such in case of negative values:

```
start, stop, step = index.indices(len(self))
```

1.0.7 Hashing

Hashing is a means to distill an object's meaning down to a single integer. Mutable objects shouldn't be hashable because their meaning can change quite easily.

```
[25]: hash('hello world')
```

```
[25]: 1294980323861224649
```

```
[26]: hash(['hello world'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[26], line 1
----> 1 hash(['hello world'])

TypeError: unhashable type: 'list'
```

Hashable objects implement the dunder `__hash__(self)`, which returns a single integer that accounts for all of its data. Often times this means combining all of an objects attributes into a tuple and hashing said tuple.

It's also good practice to implement equivalence for hashables, because we generally want two things that are equivalent to have the same hash and things with different hashes to not be equivalent.

1.0.8 Comparison

Between two objects we may compare equality with `==` or `!=` and identity with `is` or `is not`.

Equality asks "do these two objects contain the same thing" whereas identity asks "are these objects literally the same object". `a is b` can be thought of as `id(a) == id(b)`.

```
[28]: a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(id(a), id(b), id(c))
print(a is b)
print(a is c)
print(a == b)
print(a == b)
```

```
4517190272 4517194816 4517190272
False
True
True
True
```

The other relational comparison operators which evaluate to booleans: `<`, `>`, `<=`, `>=`

1.0.9 Implementing comparisons

For comparisons of the form `self comparator other`, we can implement the following methods:

- `__lt__(self, other)` : $<$ (less than)

- `__gt__(self, other)` : $>$ (greater than)
- `__le__(self, other)` : $\leq$ (less than or equal to)
- `__ge__(self, other)` : $\geq$ (greater than or equal to)
- `__eq__(self, other)` : $=$ (equivalent)
- `__neq__(self, other)` : $\neq$ (not equivalent)
- And the operator `is` cannot be overriden

Conditions that are the complement can automatically be implied, for example implementing `__eq__` means that `__neq__` will be the opposite unless otherwise specified. Implementing $<$ will also handle $>$, and $<=$ will handle $>=$, however `lt` and `eq` does not imply `le`.

16-decorators

March 21, 2024

1 16. Decorators

Decorators are a tool in python that allow you to extend the functionality of certain functions/classes