01-modules-namespaces

March 15, 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]:
```python
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]:
```python
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]:
```python
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]:
```python
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 15, 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 15, 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]:
```python
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]:
```python
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]:
```python
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 15, 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]:
```python
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]:
```python
with SomeContextManager('hello world') as x:
    print(x.value)
```

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

[13]:
```python
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 15, 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: $O(n \log^2 n)$

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 15, 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
```

```python
    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]:
```python
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 15, 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.

16-decorators

March 15, 2024

1 16. Decorators

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