IDIOMATIC PYTHON

Day 26: Leveraging the Standard Library

Welcome to day 26 of the 30 Days of Python series! Today we're going to be looking at some more of the great tools available to us in the Python Standard Library. We're also going to see how we might have used those tools in some of the code we've written thus far in the series.

I'd encourage you to explore the Standard Library in more depth after reading this post, because it contains a wealth of handy tools that save us a lot of time and effort.

The `namedtuple` function

`namedtuple` is a function available to us in the `collections` module that gives us the ability to define special tuples with named fields.

These special tuples provide awesome readability improvements, because we can retrieve values using the specified field names, much like how we use dictionary keys. We can also create instances of these tuples using keyword arguments, which allows us to give context to the data when populating a tuple with data.

In order to make use of these special tuples, we first need to define templates that specify a name for a given tuple configuration, and details the tuple's fields. This is where the `namedtuple` function comes in. `namedtuple` is used to create this template.

For example, let's say we want to create a new tuple for storing information about a book. We might define the template like this:

```python
from collections import namedtuple

Book = namedtuple("Book", ["title", "author", "year"])
```

Here we've defined a `"Book"` tuple, which has the fields `"title"`, `"author"`, and `"year"`.

We've assigned this template to the variable name `Book`, and we can create instances of this tuple like this:

```python
from collections import namedtuple

Book = namedtuple("Book", ["title", "author", "year"])

book = Book("The Colour of Magic", "Terry Pratchett", 1983)
```

As I mentioned previously, we can also use keyword arguments to give some context to the values.

```
from collections import namedtuple

Book = namedtuple("Book", ["title", "author", "year"])

book = Book(title="The Colour of Magic", author="Terry Pratch
ett", year=1983)
```

Retrieving values from these special tuples is done using dot notation, but we can also access elements by index, as usual.

```
from collections import namedtuple

Book = namedtuple("Book", ["title", "author", "year"])

book = Book(title="The Colour of Magic", author="Terry Pratch
ett", year=1983)
print(f"{book.title} ({book.year}), by {book.author}")
```

A good place to make use of `namedtuple` might have been when working with the iris data in day 14.

Our final solution looked something like this:

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        sepal_length, sepal_width, petal_length, petal_width,
species = row.strip().split(",")

        irises.append({
                "sepal_length": sepal_length,
                "sepal_width": sepal_width,
                "petal_length": petal_length,

                "petal_width": petal_width,
                "species": species
        })
```

This was perfectly fine, but it would be nice to have some protection against the data being mutated, which tuples provide. Instead of using a dictionary, we could have created an `Iris` tuple type to store all the data for a given iris.

```
from collections import namedtuple

Iris = namedtuple("Iris", ["sepal_length", "sepal_width", "pe
tal_length", "petal_width", "species"])

with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        iris = Iris(*row.strip().split(","))
        irises.append(iris)
```

The tuples we defined with `namedtuple` also have a method called `._make` which works as an alternative to using the `*` .

```
for row in iris_data[1:]:
        iris = Iris._make(row.strip().split(","))
        irises.append(iris)
```

I'd strongly recommend you check out the documentation to learn more about `namedtuple` . We should be using it all the time.

Style note

While not required, I'd strongly recommend you name your `namedtuple` template the same as the variable you assign the template to. This is really important for readability.

For example, it's perfectly legal to do this:

```python
Movie = namedtuple("Film", ["title", "director", "year"])
```

But you should write either of these instead:

```python
Film = namedtuple("Film", ["title", "director", "year"])
```

```python
Movie = namedtuple("Movie", ["title", "director", "year"])
```

The `partial` function

The `partial` function is a way to create a new version of a function, where some portion of the arguments are already given.

For example, let's look at our `exponentiate` function from the day 13 exercises.

```python
def exponentiate(base, exponent):
        return base ** exponent
```

This is basically a reimplementation of the built in `pow` , and it works perfectly well. But let's say that most of the time we want to use this to square or cube the number passed in as the `base` . It's pretty annoying having to write out `exponentiate(5, 2)` when we could be writing something like `square(5)` . It's also less readable.

While we could go ahead and write another function like this:

```python
def square(base):
	return base ** 2
```

This is not a very reasonable solution if we were dealing with a more complicated function. This would lead to a lot of duplicate code.

Instead, we can use the `partial` to create a new function which has some fixed value for the `exponent` parameter.

Let's use `partial` to create a `square` function and a `cube` function from our original `exponentiate` function.

```python
from functools import partial

def exponentiate(base, exponent):
	return base ** exponent

square = partial(exponentiate, exponent=2)
cube = partial(exponentiate, exponent=3)
```

Just like that, we have a couple of new functions we can call:

```python
from functools import partial

def exponentiate(base, exponent):
	return base ** exponent

square = partial(exponentiate, exponent=2)
cube = partial(exponentiate, exponent=3)

print(square(4))  # 16
print(cube(5))    # 125
```

You can find more information on `partial` here

You can find more information on `partial` here.

Note

One thing you do have to be a little bit careful of is the order of your parameters.

If we wanted to create functions that set a fixed value for `base` using a keyword argument, that would be problematic, because the first positional argument we pass into our new function would also be assigned to `base`. We would therefore need to pass in the value for `base` as a positional argument when creating our `partial` function.

The `defaultdict` type

The `collections` module has a few special types of dictionaries for us to work with. The `defaultdict` type is a dictionary that lets us specify some default value to return when we attempt to access a key which doesn't exist.

This can be very helpful, because it saves us having to call `get`, specifying a default value in each case. It also means we can often do away with a lot of logic to check what came back from calling `get`.

For example, let's say we have a dictionary of users like this, where each key is a user id:

```python
from collections import namedtuple

User = namedtuple("User", ["name", "username", "location"])

users = {
        "0001": User("Phil", "pbest", "Hungary"),
        "0002": User("Jose", "jslvtr", "Scotland"),
        "0003": User("Luka", "lukamiliv", "Serbia")
}
```

This is a good opportunity to get in some more practice using `namedtuple`, so I've made each of the values a `User` tuple.

Now let's say we want to retrieve a user from this dictionary using their id, and if a user can't be found, we print a message to the console.

We'd probably end up doing something like this:

```
from collections import namedtuple

User = namedtuple("User", ["name", "username", "location"])

users = {
        "0001": User("Phil", "pbest", "Hungary"),
        "0002": User("Jose", "jslvtr", "Scotland"),
        "0003": User("Luka", "lukamiliv", "Serbia")
}

user_id = input("Please enter a user id: ")
user = users.get(user_id)

if user:
        print(user)
else:
        print("Could not find a user matching that user id.")
```

Now let's rewrite this using a `defaultdict`.

In order to specify a default value to return when a key cannot be found, we need to use a function. In this case I'm going to use a lambda expression to define this function, and it's going to return a simple string.

The `defaultdict` is going to call this function any time a missing key is requested, and the return value of that function will be returned.

```
from collections import defaultdict, namedtuple

User = namedtuple("User", ["name", "username", "location"])

users = defaultdict(
        lambda: "Could not find a user matching that user i
d.",
        {
                "0001": User("Phil", "pbest", "Hungary"),
                "0002": User("Jose", "jslvtr", "Scotland"),
                "0003": User("Luka", "lukamiliv", "Serbia")
        }
)

user_id = input("Please enter a user id: ")
print(users[user_id])
```

The documentation has a lot of interesting examples of innovative ways to use a `defaultdict`. One way is as a sort of counter.

To demonstrate this, let's say we're trying to keep track of an inventory for a character in an RPG of some kind. We're using a dictionary to store what is in the character's inventory, with the keys being the item names, and the values being a count of how many of this item the character has.

I'm also going to create a function which is going to modify this dictionary, allowing the user to add new items.

```
inventory = {}

def add_item(item, amount):
        if inventory.get(item):
                inventory[item] += amount
        else:
                inventory[item] = amount
        inventory[item] = amount

add_item("bow", 1)
add_item("arrow", 20)
add_item("arrow", 20)
add_item("bracer", 2)

print(inventory)  # {'bow': 1, 'arrow': 40, 'bracer': 2}
```

This works, but we can do a better job by using a `defaultdict` with `int` as the factory function.

When `int` is called without any arguments, it returns `0`. This is very useful for us, because it means when we do this:

```
inventory[item] += amount
```

Which is just the same as this:

```
inventory[item] = inventory[item] + amount
```

The right hand `inventory[item]` is going to be replaced with `0`, allowing us to create a new key with a starting value equal to the amount of that item added. Pretty neat.

```
from collections import defaultdict

inventory = defaultdict(int)

def add_item(item, amount):
        inventory[item] += amount

add_item("bow", 1)
add_item("arrow", 20)
add_item("arrow", 20)
add_item("bracer", 2)
add_item("bracer", 2)

print(inventory)  # defaultdict(<class 'int'>, {'bow': 1, 'arrow': 40, 'bracer': 2})
```

As you can see, this led to a marked simplification of our code.

Exercises

1) Define a `Movie` tuple using `namedtuple` that accepts a title, a director, a release year, and a budget. Prompt the user to provide information for each of these fields and create an instance of the `Movie` tuple you defined.

2) Use a `defaultdict` to store a count for each character that appears in a given string. Print the most common character in this dictionary.

3) Use the `mul` function in the `operator` module to create a `partial` called `double` that always provides `2` as the first argument.

4) Create a `read` function using a `partial` that opens a file in read (`"r"`) mode.

You can find our solutions to the exercises here.