teclado

IDIOMATIC PYTHON

Day 25: Writing Idiomatic Python

Welcome to day 25 of the 30 Days of Python series! Today we're going to be learning some more common patterns and tricks used by professional Python developers.

This is going to help us build on the idioms we've already learnt, so that we can make the jump to writing really professional looking code!

Pythonic code

Before we dive into looking at any specific patterns, I just want to take a moment to talk about "Pythonic code". This is a term you're likely to hear a fair bit as you continue learning about Python.

There's a lot of confusion out there about what exactly "Pythonic" code is, but the term is actually fairly easy to define. Python code is said to be Pythonic when it's written in a way that adopts the idioms of the Python language, and which conforms to a certain set of stylistic principles.

An **idiom** is a language, dialect, or style of speaking peculiar to a people.

There's a fairly intuitive analogy with natural language here.

Let's look at English as an example. In English, we have the word, "goodbye", but native speakers of English rarely say, "goodbye". Instead, we favour things like, "cya", "bye", and "catch you later". Goodbye generally sounds a bit awkward and stilted in most situations.

The way we write our Python code is very much the same. There are ways of writing Python that make sense, and are perfectly correct in a functional sense, but which don't look quite right to Python "natives".

Writing Pythonic code is about writing code which *does* look right to these other Python developers.

With that in mind, let's learn about some of the things which will make us look more like true Pythonistas.

Truth values as conditions

This is something we spoke about briefly back in day 5, but I think it's really worth reinforcing this pattern. We're also going to be building on this pattern in the following sections.

As a quick reminder, in Python every value has an associated truth value which determines whether it evaluates to `True` or `False` when we pass it to the `bool` function.

In general, values in Python evaluate to `True`, barring a small number of exceptions.

One important category of exceptions are empty sequences and collections, which includes things like strings, lists, `range` objects, dictionaries, etc.

The fact that every value can be considered as `True` or `False` gives us a lot of power, because we can use any value as a condition for things like `while` loops, or conditional statements.

For this reason, we generally never need to write code like this:

```
my_list = []

if len(my_list) == 0:
	print("The list is empty")
else:
	values = ', '.join(str(value) for value in my_list)
	print("The list contains: {values}")
```

Instead, the more Pythonic approach would be to test the truth value of `my_list`. If the list is empty, it will evaluate to `False`, so we can use this to control the flow of the conditional statement.

```
my_list = []

if my_list:
	values = ', '.join(str(value) for value in my_list)
	print("The list contains: {values}")
else:
	print("The list is empty")
```

This approach is also very useful for checking whether or not a function or method returned `None`.

```
from operator import add, mul, sub, truediv
```

```
operations = {
        "add": add,
        "divide": truediv,
        "multiply": mul,
        "subtract": sub
}

selected_option = input("Select the operation to perform: ").
strip().lower()

operation = operations.get(selected_option)  # returns None i
f the key is invalid

if operation:
        a = input("Please enter a value for the first operan
d: ")
        b = input("Please enter a value for the second operan
d: ")

        print(operation(a, b))
else:
        print("Invalid operation")
```

Note

There are situations where we have to check explicitly for some values instead, so be judicious about when to use this structure.

For example, if `None` is a value we're looking for specifically, we should check if the value `is None` or `is not None`, rather than checking truth values. This is because `0`, `False`, `None`, and many other are all going to appear the same from the perspective of truth values.

Boolean operators for control flow

If you've been reading the additional resources, you've likely already encountered the `and` and `or` operators, but they're not something we've really covered in the series.

Their basic use in Python is to combine expressions to make compound conditions.

For example, we can do something like this:

```
x = int(input("Please enter a number between 1 and 10: "))

if x > 0 and x < 11:
        print(x)
else:
        print("Invalid selection")
```

Here is the same example written with `or` :

```
x = int(input("Please enter a number between 1 and 10: "))

if x < 1 or x > 10:
        print("Invalid selection")
else:
        print(x)
```

In this sense, the operators are fairly straightforward. For an `and` condition to evaluate to truthy, both operands must be truthy. For an `or` condition, it's enough that either of the operands are truthy.

However, in Python `and` and `or` have a very interesting property. They don't evaluate to `True` and `False` : they instead evaluate to one of the operands.

In the case of `and` , we get the first operand back if that operand is falsy; otherwise, we get the second operand. Let's consider a few examples to see how this works.

```
4 and 0               # evaluates to 0
```

```
4 and 0              # evaluates to 0
[] and 3             # evaluates to []

True and False       # evaluates to False
True and True        # evaluates to True
4 and 5              # evaluates to 5
```

If we use `and` as a condition in something like a conditional statement, this functions exactly like if `and` evaluated to `True` or `False`.

In cases where the the first operand is falsy, we already know that the expression can't be truthy, so we just give back the falsy values. This in turn is then evaluated to `False`.

If the first operand is truthy, we instead give back the second operand without bothering to check it. Why?

Because if it's truthy, then both values were truthy, and the truthy value we returned will evaluate to `True`. If the second operand is instead falsy, then both values weren't truthy, so when the falsy value evaluates to `False`, we get exactly what we want. We don't need to check this second value at all.

`or` works in much the same way, but it returns the first operand if it's truthy, otherwise it returns the second operand. Here are a few examples:

```
4 or 0               # evaluates to 4
[] or 3              # evaluates to 3
False or None        # evaluates to None
True or False        # evaluates to True
4 or 5               # evaluates to 4
```

Let's look at some ways we can use this property.

Using `or` to replace falsy values

A common way to use `or` is to replace values which are falsy. For example, let's consider a case where we want to enter a default name when the user fails to provide one.

Using the techniques we learnt in the last section, we might be tempted do something like this:

```
name = input("Please enter your name: ").strip().title()

if not name:
        name = "John Doe"
```

However, we can use the `or` operator to cut this down to a single line like this:

```
name = input("Please enter your name: ").strip().title() or
"John Doe"
```

Here we're saying assign the user input to `name` if it was truthy (i.e. not an empty string); otherwise, assign the string, `"John Doe"` .

Using `and` to change how we handle return values

Using `and` is a lot less common, but there are some interesting things we can do with this operator as well.

Let's say we have a function like this:

```
def divide(a, b):
        try:
                return a / b
        except ZeroDivisionError:
                return
```

Here we have two possible types of return value. We either get a float representing the result of the division, or we get `None` .

Let's say I want to test if the result of some division is an integer, using the `is_integer` method. I then want to inform the user whether or not the number is an integer.

Using conditional statements, we'd have to do something like this:

```
def divide(a, b):
        try:
                return a / b
        except ZeroDivisionError:
                return

a = 6
b = 0

result = divide(a, b)

if result:
        if result.is_integer():
                print(f"{a} / {b} produces an integer resul
t.")
```

Using `and` , we can instead write something like this:

```
def divide(a, b):
        try:
                return a / b
        except ZeroDivisionError:
                return

a = 6
b = 0

result = divide(a, b)

if result and result.is_integer():
    print(f"{a} / {b} produces an integer result.")
```

Style note

While I think it's an important one to know about, I wouldn't necessarily suggest you use this `and` approach, and I certainly wouldn't recommend you use it all the time. It has a far more limited use case than `or`.

Once you understand it, it's relatively easy to follow what's going on, but it's particularly opaque to people who aren't familiar with the idiom, and I think that often makes it a worse choice than using some combination of conditional statements and `try` statements.

Truncation over slicing

Let's say we have a list, and we want to discard some values from that list. An example might be a case where we've somehow filtered a list, but we only want the first ten values.

An approach using slicing might look like this:

```
numbers = [1, 54, 2, -4, -65, 23, 97, 45, 14, 19, 73, -6, 31, 92, 3]

positive_numbers = sorted(number for number in numbers if number > 0)
positive_numbers = positive_numbers[:10]

print(positive_numbers)  # [1, 2, 3, 14, 19, 23, 31, 45, 54, 73]
```

This approach is quite verbose, and there's a lot of opportunity for typing errors here, because we're having to type the same variable name twice. If we had a few similar names, this could easily lead to a bug where we reference two different variables by mistake.

Incidentally, this is one reason to also prefer this:

```
i += 1
```

To this:

```
i = i + 1
```

It's also not terribly efficient, because slicing a list like this creates a brand new list. Instead, a better approach would be to delete items from the original list. We can do this like so:

```
numbers = [1, 54, 2, -4, -65, 23, 97, 45, 14, 19, 73, -6, 31,
92, 3]

positive_numbers = sorted(number for number in numbers if num
ber > 0)
del positive_numbers[10:]

print(positive_numbers)  # [1, 2, 3, 14, 19, 23, 31, 45, 54,
 73]
```

Not only is this shorter, and more efficient, it's also more readable. It makes it very clear that we're deleting items from the `positive_numbers` list, whereas the slice approach obfuscated this a bit with an additional assignment.

Of course, there are cases where slicing is the correct solution. We don't always want to destroy the original collection, after all.

Check for multiple values using `in`

One thing I often see when reviewing student code is something like this:

```
proceed = input("Would you like to continue? ").strip().lower
()

if proceed == "y" or proceed == "yes" or proceed = "continue"
:
        ...
```

This is a good idea. They're trying to catch lots of different ways the user might try to respond, so that the program is more accommodating of the users.

The approach, however, is extremely verbose, and there's a much better way of writing this kind of code.

```
proceed = input("Would you like to continue? ").strip().lower
()

if proceed in ("y", "yes", "continue"):
        ...
```

If you want to check for multiple values, using the `in` keyword is much shorter and cleaner.

Exercises

1) Write a function that prompts the user for their name and then greets them. You should process the string by removing any whitespace and converting the string to title case.

If after processing the string you're left with an empty string, the function should replace the empty string with "World" in the output.

2) Write a function to determine whether or not a string contains exclusively ASCII letters (a to z in either upper or lowercase).

Hint: You should look at the constants in the `string` module. Documentation can be found here.

3) Use the `sample` function in the `random` module to create three lists, each containing fifteen numbers from `1` to `100` (inclusive). Sort each of these lists into descending order (largest first), and then truncate each list so that only 5 items remain in each.

You can find our solutions here.