KICKING OFF YOUR LEARNING

Day 3: Formatting Strings and Processing User Input

Python guru sitting down with a screen instead of a face, and day 03 displayed on it.

← →

Welcome to day 3 of the 30 Days of Python series! In this post we're going to be talking about concatenating strings (joining strings together), string interpolation (putting values inside strings), and processing user input. We're also going to talk about how we can better document our code using comments.

If you missed day 2, I'd recommend checking it out if you're not comfortable with any of the following:

- Strings
- Defining variables
- The `input` function
- How to read an error message

I hope you're excited for this one, because these tools are going to let us do some really cool stuff!

String concatenation

I want to start by looking at an example from the exercises in the last post:

```python
hourly_wage = input("Please enter your hourly wage: ")
hours_worked = input("How many hours did you work this week? ")

print("Hourly wage:")
print(hourly_wage)

print("Hours worked:")
print(hours_worked)
```

Here we ask the user for two pieces of information, and then we print that information back to the console along with some headings.

This is all well and good, but wouldn't it be nice if we could have the `hourly_wage` value on the same line as the `"Hourly wage:"` label? This is where string concatenation (joining two or more strings together) comes in.

To concatenate things means to join or link things together, and this is what we want to do with our strings. We want to take the string, `"Hourly wage:"` , and we want to join to it the string we assigned to the variable, `hourly_wage` .

In Python, we can just use the `+` operator for this, like so:

```python
hourly_wage = input("Please enter your hourly wage: ")
hours_worked = input("How many hours did you work this week? ")
```

```
print("Hourly wage:" + hourly_wage)
print("Hours worked:" + hours_worked)
```

There’s a problem here though, much like the problem we faced when writing input prompts. The value of `hourly_wage` and `hours_worked` are going to end up directly next to the final character of the strings we're joining them to. In order to avoid this, we need to remember to add a space:

```
hourly_wage = input("Please enter your hourly wage: ")
hours_worked = input("How many hours did you work this
week? ")

print("Hourly wage: " + hourly_wage)
print("Hours worked: " + hours_worked)
```

This operation is another expression which is going to evaluate to a new string containing the characters of both strings.

Much like when we used `+` for adding together numbers, we can chain together as many strings as we like using `+`. However, we have to be very careful not to combined types.

For example, we can’t do something like this, where we try to join an integer onto a string:

```
print("Hourly wage: " + 20)
```

If we run this code, Python is going to raise an exception:

```
Traceback (most recent call last):
    File "main.py", line 1, in <module>
        print("Hourly wage: " + 20)
TypeError: can only concatenate str (not "int") to str
```

This time we get a `TypeError`, which is telling us that we’ve tried to concatenate an `int` (integer) to a `str` (string), but this operation is only supported between strings.

So the question now, is what do we do if we need to perform some arithmetic operations, and then put the result in a string? Luckily Python provides us some tools to convert values of one type to another.

Converting strings, integers, and floats

First, let's talk about converting integers and floats to strings. The tool we're going to be using is the conveniently named `str`, and we can think of it as being very much like a function.

When we call `str`, passing in an integer or a float, we're going to get a new string containing the numerals that make up the number, along with a decimal point character in the case of floats.

For example, we can write something like this:

```
age = str(28)
```

Which is going to assign to `age` the same thing as if we wrote:

```
age = "28"
```

Converting from a string to an integer or a float is a little more complicated, because not every string is a valid number. Moreover, even if a string is a valid number, it might not be a valid number of the type we're trying to convert to.

If we want to convert a string to an integer, we need to call `int`, passing in the string. The characters of the string must represent an integral number.

```
age = int("28")
```

If we try to pass a string representation of a number with a decimal component, we're going to get a `TypeError`. For example, something like this:

```
hourly_wage = int("18.50")
```

Is going to raise the following exception:

```
Traceback (most recent call last):
    File "main.py", line 1, in <module>
        int("18.50")
ValueError: invalid literal for int() with base 10: '1
8.50'
```

Instead of `int`, we'd need to call `float` here instead:

```
hourly_wage = float("18.50")
```

`float` can also handle input without a decimal component. For example, if we pass in `"20"`, we're going to get the float `20.0` back.

In addition to converting from strings to integers and floats, and *vice versa*, we can also convert between integers and floats using the same `int` and `float`. In this case, passing a float to `int` is going to work, but it's going to "truncate" the float, essentially throwing away everything after the decimal point.

Truncation is the act of shortening something by removing some part of it. For example, we can refer to truncating a float, which means that we're removing everything after the decimal point.

String interpolation with the `format` method

String concatenation is a useful thing to know about, but it's a bit clunky, and it's prone to spacing errors, as we've seen. A better approach is often to use string interpolation instead, especially when working with content that has a sentence-like structure.

String interpolation is the act of inserting some new content into an existing string.

Unlike concatenation, interpolation relies on us putting placeholders in a string which we can fill with values. This makes it a lot harder to fall prey to formatting problems, because the placeholders are in the exact position the values will occupy. Another great benefit of interpolation: we don't have to worry about types! Python will ensure we get a string representation of all of our values.

There are several ways we can perform string interpolation in Python, but first we're going to look at using the `format` method.

When using the `format` method, the placeholders for our values are curly braces, which look like this: `{}`. For example, let's say we want to write a string like:

```
"John is 24 years old!"
```

But instead of just `"John"` and `"24"`, we want to be able to write any name and any age here. All we have to do accomplish this is replace `"John"` and `"24"` with curly braces, like so:

```
"{} is {} years old!"
```

This is a perfectly valid string, and we can even print it in its current form. We can also assign it to a variable if we want, which can often be very useful.

In either case, in order to fill in the values, we need to call the `format` method on the string with our placeholders, which is going to yield a new string. We do this using dot notation like this:

```
"{} is {} years old!".format()
```

We're not quite done yet though, because `format` needs to know which values to put in the string. There are actually a few ways we can do this, but the simplest is just to pass format one expression for each placeholder in the order we want them placed.

For example we can do this:

```
"{} is {} years old!".format("John", 24)
```

`format` gives us a new string, and the resulting string is going to contain this:

```
"John is 24 years old!"
```

Alternative ways of using `format`

Often just passing in values in order is enough, but we have other options at our disposal.

For example, let's say we have a string like this:

```
"{} is {} years old, and {} works as a {}."
```

The intention here is for our output to look something like this:

```
"John is 24 years old, and John works as a web develop
er"
```

Here we're using the name in two different places. Using our existing knowledge, we can pass in a value for each placeholder in order:

```
output = "{} is {} years old, and {} works as a {}."
```

```python
print(output.format("John", 24, "John", "web develope
r"))
```

However, this is a bit clunky. Instead of doing this, we can number our placeholders, which will let us reuse values.

One thing to be aware of is that we start counting at `0` when we're programming, so the first value we pass in is value `0`, not value `1`.

With that in mind, we rewrite the code above like so:

```python
output = "{0} is {1} years old, and {0} works as a
 {2}."

print(output.format("John", 24, "web developer"))
```

This is much better, especially if we're using the same values over and over again. However, an even better approach is often to use names for our placeholders. This works a lot like using variables.

We start by writing some name inside each placeholder like so:

```python
output = "{name} is {age} years old, and {name} works
 as a {job}."
```

Now we can call the `format` method like this:

```python
print(output.format(name="John", age=24, job="web deve
loper"))
```

What we're saying here is that we want `name` to be replaced with `"John"`; `age` to be replaced with `24`; and `job` to be replaced with `"web developer"`.

We're going to be looking at what's actually going on in a lot more detail later in the series.

String interpolation with f-strings

The `format` method is really useful, but in Python 3.6 we got a new piece of syntax called f-strings which makes string interpolation a lot easier in many cases.

Let's look at an example first. Here is one of our earlier examples using `format`:

```
name = "John"
age = 24

"{} is {} years old!".format(name, age)
```

And here is the same example using the newer f-string syntax:

```
name = "John"
age = 24

f"{name} is {age} years old!"
```

The first thing of note is this `f` directly before the string we're inserting our values into. This is where f-strings get their name, and it tells Python that we want to format the following string.

The other interesting thing is that we've referred to values directly inside the placeholders. This isn't like with `format`, where we named our placeholders: we're directly referencing values inside the curly braces.

We can actually write any expression we want here. For example, let's calculate John's age in months right here in the string:

```
name = "John"
age = 24
```

```python
f"{name} is {age * 12} months old!"
```

In case you were wondering, f-strings are also expressions, so we can assign them to names, or we can print them if we want.

```python
name = "John"
age = 24

print(f"{name} is {age * 12} months old!")
```

Comments

Using good, descriptive names we can do a lot to make our code self-documenting; however, sometimes even great code can be hard to reason about. Some things are just complicated, after all.

In instances like this, it can be useful to use comments to explain to readers of our code (which includes us) what the code is doing.

In order to write a comment, we just need to put a `#` directly before the message we want to write. This is going to tell Python not to treat what follows as code.

```python
name = "John"
age = 24

# prints the person's name and age in months
print(f"{name} is {age * 12} months old!")
```

Try not to go overboard with comments, as having comments on every other line can cause more harm than good when it comes to readability.

Basic String Processing

The final thing I want to mention in this post is some common string processing operations, such as changing the case of the letters, or removing white space (space characters, tabs, newlines, etc.) from the

ends of strings. This kind of thing happens all the time when we're dealing with user input.
We have a number of different options for changing the case of letters in a string. Here we're going to focus on four important options: `lower` , `upper` , `capitalize` , and `title` .

`lower` and `upper` turn the entire string to lowercase and uppercase, respectively. Characters which don't have a case, such a punctuation characters, are ignored.

`capitalize` is going to turn the first character to uppercase, with the rest being lowercase. `title` is going to turn the string to title case, which means every word starts with a capital letter, and all other letters are turns to lowercase.

In order to use these methods, we just need to use the dot notation again, just like with `format` .

```
"Hello, World!".lower()       # "hello, world!"
"Hello, World!".upper()       # "HELLO, WORLD!"
"Hello, World!".capitalize()  # "Hello, world!"
"hello, world!".title()       # "Hello, World!"
```

In each case, we're writing an expression, and the value of that expression is a new string in the case requested.

There's not much more to it when it comes to changing case, but what about removing white space from the ends of a string? For that, we can use the `strip` method.

`strip` is actually a lot more versatile than this, and we can have it remove anything we like, but by default it will give us a shiny new string with any extraneous white space removed from both ends.

We use it in exactly the same way as `lower` , `upper` , etc.

```
"   Hello, World!   ".strip()  # "Hello, World!"
```

Processing multiple times

Processing multiple times

We can apply multiple processing methods to a string. Let me show you two ways to do it.

The first way involves creating a variable and re-assigning to it for every method call, like so:

```
user_name = "  ROLF SMITH   "
user_name = user_name.strip()   # "ROLF SMITH"
user_name = user_name.title()   # "Rolf Smith"
```

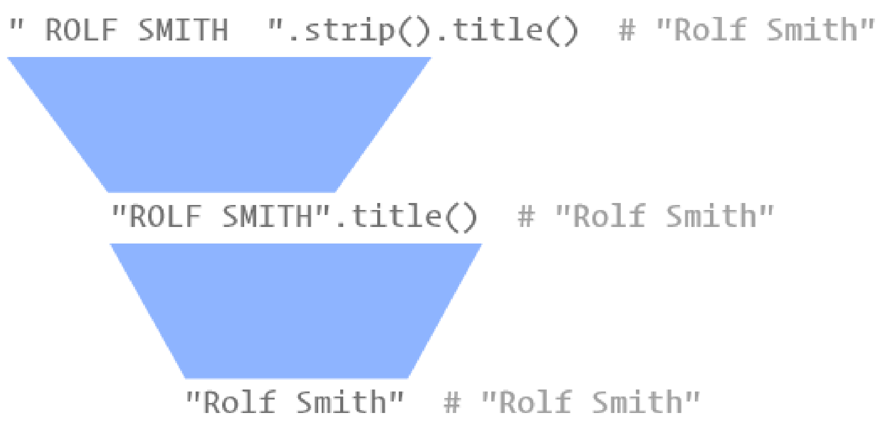
Although this is very verbose, and re-assigning to a variable multiple times generally feels suboptimal and inefficient (although it's not *slow* by any stretch, Python can re-assign to variables very quickly).

So the second way to do this involves using the result of each step *inline*, like so:

```
user_name = "  ROLF SMITH   ".strip().title()
```

We'll learn more about exactly how this works when we look at functions and return values, but for now treat each method call, such as `" ROLF SMITH ".strip()` , as the string that it would give you back.

Therefore even on inline calls like that, these three are equivalent:



```
"  ROLF SMITH   ".strip().title()   # "Rolf Smith"
"ROLF SMITH".title()   # "Rolf Smith"
"Rolf Smith"   # "Rolf Smith"
```

Exercises

1. Using the variable below, print `"Hello, world!"` .

```
greeting = "Hello, world"
```

You can add the missing exclamation mark using string concatenation, `format` , or f-strings. The choice is yours.

2. Ask the user for their name, and then greet the user, using their name as part of the greeting. The name should be in title case, and shouldn't be surrounded by any excess white space.

For example, if the user enters `"lewis "` , your output should be something like this:

```
Hello, Lewis!
```

3. Concatenate the string `"I am "` and the integer `29` to produce a string which reads `"I am 29"` .

Remember that we can only concatenate strings to other strings, so you will have to convert the integer to a string before you can perform the concatenation.

4. Format and print the information below using string interpolation:

```
title = "Joker"
director = "Todd Phillips"
release_year = 2019
```

The output should look like this:

```
Joker (2019), directed by Todd Phillips
```

Once you're done with the exercises, make sure to check your answers against our solutions. If you have an alternative solution, feel free to share it on our Discord server!

Project

Once you've finished the exercises above, you should try the first of our mini projects.

Additional Resources

There are a great deal of string methods available to us in addition to the ones we've looked at here. You can find a comprehensive list in the documentation.

Don't feel like you have to memorise all of these. You can always look up method names if you forget them, and the documentation describes how to use them.

If you're interested, we also have a post covering some additional options for when using `print` . The second half of the post gets into working with files, which I'd recommend skipping over for now.