KICKING OFF YOUR LEARNING

Day 2: Strings, Variables, and Getting Input from Users

Welcome to day 2 of the 30 Days of Python series! In this post we're going to be talking about how to use strings, getting user input, and naming values in our programs using variables.

If you missed day 1, I'd recommend checking it out if you're not comfortable with any of the following:

- Integers
- Floats
- Simple arithmetic operators (`+` , `-` , `/` , and `*`)
- The `print` function
- What expressions are

Without further ado, let's talk about strings.

String basics

Strings in Python are ordered sequences of zero or more characters, and we can use them to represent arbitrary collections of symbols. This might be words, whole sentences, or random strings of letters, numerals, and punctuation. Strings are sometimes called "string literals".

In order to create a string, we just need to wrap some series of characters in quotation marks:

```
"This is a string!"
```

We can use single or double quotes. They don't have any different meaning in Python, and there's no strong reason you should exclusively use one or the other.

```
'This is also a valid string!'
```

We can't, however, start a string with one type of quote and end the string with a different type. This is invalid syntax:

```
'This is not valid string!"
```

Printing strings works just like it does for numbers. We can just pass in a string when calling the `print` function, like so:

```
print("Hello, world!")
```

As I said in the opening paragraph, strings are a sequence of zero or more characters, and we can define an empty string by writing a pair of quote with nothing in between:

```
""
```

or

```
''
```

Note that this is not the same as the following:

```
" "
```

That's a string containing a single space, and spaces are characters, so the string is **not** empty.

Naming values

One vital tool that we're missing at the moment is a way to refer to values that we've calculated. Remember that we can write expressions like `45 + 56`, and Python does this calculation, even though we haven't printed anything to the console. The question is, how do we refer to the result of that expression?

As it happens, we can name values in Python using an *assignment operation*. The syntax for binding a value to a name is relatively simple. We just need to write the name we want to use, an `=` symbol, and the some expression. The result of the expression—the value it evaluates to—is what the name refers to. This name is called a variable.

Here are a couple of examples:

```
name = "Phillip Best"
age = 28
```

The variable names we choose are entirely up to us, but there are a few rules we need to keep in mind:

- Variable names can include letters, numbers, and underscore

(_) characters.

- Variable names can't start with a number, though starting with an underscore is allowed.
- Variable names are case sensitive, but can be in any case.

These limitations mean that all of the following variable names are invalid:

```
1st_value
given-name
john's_age
example variable
```

- `1st_value` isn't allowed because it starts with a number.
- `given-name` contains an illegal hyphen character.
- `john's_age` isn't allowed because of the apostrophe.
- `example variable` includes a space, which also isn't allowed.

These rules make a lot of sense. For example, if numbers were allowed as the first character of a variable name, we'd be able to use single character names like `4`. However, this is also how we create an integer and could lead to confusion.

Similarly, `given-name` contains a `-` character, which conflicts with the `-` operator for subtraction, and `john's_age` has a similar issue with the `'`, which is used to define strings.

One valid variable name which might surprise you is `_`. This has a special use case which we'll touch on later in the series.

If we want to define variable names with multiple words, we can separate those words with an underscore, like so:

```
given_name = "Phillip"
surname = "Best"
```

Style note

Style note

While we have a lot of freedom in how we define our variable names, there are some conventions that we should keep in mind to help keep our code as readable as possible.

The first is that we should use descriptive names for your variables. It's perfectly legal to write the following:

```
x = "Phillip"
y = "Best"
```

But the problem is, `x` and `y` don't really mean anything. Compare this to `given_name` and `surname`, which immediately tell us what the values they refer to are. Using variables should never make your code harder to understand.

Another thing which you may have noticed is that all of my variable names have been lower case. There are instances where we use capital letters for names in Python, but when we do this, it has special meaning. Until we get to those use cases, avoid using capital letters, because it's going to give misleading information to developers familiar with Python's naming conventions.

Referring to variables

Now that we know how to bind values to names, the next step is learning how to retrieve those values from our variables. The good news is that this is extremely simple. All we have to do is write the variable name where we want to use the value.

For example, we can write something like this:

```
hourly_wage = 20.00
hours_worked = 40

print(hourly_wage * hours_worked)
```

Here we've defined two variables `hourly_wage`, which is a float, and `hours_worked` which is an integer. We've then referred to the values and used them as operands for the `*` operator.

If we run the code above, we should see `800.00` printed to the console. Remember that multiplying any number by a float results in a float.

As it happens, referring to a variable name is another type of expression. When we write a variable name anywhere in our code, it evaluates to the value we bound to that name.

But what happens if we haven't bound anything to that name yet? Let's have a look!

The exception

Let's go over to `repl.it` and we'll write a program containing only a single expression, where we refer to a variable we haven't yet defined:

```
name
```

If we run this program, we're going to get some angry red text in the console, which looks something like this:

```
Traceback (most recent call last):
        File "main.py", line 1, in <module>
                name
NameError: name 'name' is not defined
```

This traceback and the other information provided here is extremely useful, and I highly recommend you get in the habit of reading it in detail when something goes wrong. With that in mind, I think it's worth spending a little bit of time to learn how to read this.

Definition: A traceback is a report that gives us information about what was happening in our code when an exception occured.

The first part is the traceback itself, which is going to tell us *where* the exception occurred. As a side note, an exception is usually an error, but there are cases where exceptions are used for other purposes in Python as well.

The traceback tells us that this exception occurred in a file called `main.py`. If you're on `repl.it`, remember that this is the file that always get run when we press the "Run" button. It also tells us that the exception occurred on line 1. On the next line, it tells us what the problematic code is. In our case, this is the reference to the variable, `name`.

This is all good information to know, and it lets us very quickly pinpoint where in our code we should be looking for the problem.

Under the traceback, we have the name of the exception that occurred (a `NameError` in our case), along with some message explaining the exception in more detail. As we can see, the problem is that `name` is not defined. In other words, we haven't told Python what `name` means in our code.

There's another situation where a `NameError` might occur, so let's take a look at that:

```
print(hourly_wage * hours_worked)

hourly_wage = 20.00
hours_worked = 40
```

Here we've defined our two variable names, but if we run the code, we're going to get the following message in the console:

```
Traceback (most recent call last):
    File "main.py", line 1, in <module>
```

```
        print(hourly_wage * hours_worked)
NameError: name 'hourly_wage' is not defined
```

Once again we have an exception on line 1 of the `main.py` file, and once again the exception is a `NameError`. In this case, it's telling us that `hourly_wage` is not defined. The actual problem is that both `hourly_wage` **and** `hours_worked` are undefined, but `hourly_wage` was encountered first.

The reason we're getting this error is because Python is running our code top to bottom. When it encounters `print(hourly_wage * hours_worked)` on line 1, it doesn't yet know that we've defined `hourly_wage` and `hours_worked` later in the code. For this reason, we have to make sure to always define our variables before we use them in the code. If we don't, Python is going to raise a `NameError` exception, just like we've seen in these examples.

Getting values from the user

So far we've provided all the values in our Python code, but sometimes we need to get values from the user instead. For example, maybe we want to get the user's name and age.

Luckily we don't have to implement something like this from scratch, because Python comes with another function to do this exact job. This function is called `input`.

We call `input` in exactly the same way as `print`, we just need to write the function name and put opening and closing parentheses directly after:

```
input()
```

If you run the code above, you won't see anything in the console, but if you look at the "Run" button in repl.it, you'll see that it says "Stop". This means the program is still running. You'll also find that

you can type in the console.

If you press enter, the program will end. That's because when you press enter, the `input` function is going to stop accepting user input. Since we have no other code in our program, Python then runs out of code to execute, and terminates the program.

While we can call `input` by just writing `input()`, usually we're going to want to provide some kind of prompt, for the user to know what we want from them. Your first instinct might be to do something like this:

```
print("Please enter your name:")
input()
```

That'll certainly work, but `input` is actually capable of providing its own prompt. We just need to write the desired prompt inside the parentheses when we call the function:

```
input("Please enter your name:")
```

I'd encourage you to run that line of code for yourself, because there's a bit of a problem. At the moment, the user is writing directly next to the colon I added at the end of the prompt. In order to avoid this, make sure you put a space or two at the end of your prompts. It's going to make a big difference to the user experience.

```
input("Please enter your name: ")
```

The final piece of the puzzle is storing this value somewhere. Remember that function calls are expressions, and therefore evaluate to some value. The `input` function evaluates to a string which contains whatever the user wrote in response to the prompt. We can therefore assign this string to a variable if we want, like so:

```
name = input("Please enter your name: ")
```

If the user enters `John`, `name` will end up containing the name, `"John"` as a string.

Exercises

Now that we've covered how to use strings, variables, and the `input` function, it's time to practice with some exercises.

1. Ask the user for their name and age, assign theses values to two variables, and then print them.
2. Investigate what happens when you try to assign a value to a variable that you've already defined. Try printing the variable before and after you reuse the name.
3. Below you'll find some code with a number of errors. Try to go through the program line by line and fix the issues in the code. I'd encourage you to try running the program while you're working on it, as reading the error messages is great practice for debugging your own programs.

```
hourly_wage = input("Please enter your hourly wage: ')

prnt("Hourly wage: ")
print(hourlywage)
print("Hours worked: ")
print(hours_worked)

hours_worked = input("How many hours did you work this
week? ")
```

You can find solutions to today's exercises here.

I hope this hasn't been too challenging so far, but if you're stuck, feel free to join us on our Discord server and ask us any questions. We'd be more than happy to help!

Next time we're going to be looking at processing user input, and we'll be diving a little deeper into strings. Hope to see you there!