

Values and Variables

Values and Variables



The plan

In this module, we will cover

- How to **assign** a **value** to a **variable**.
- The kinds of basic values **types** there are.
- Some basic **operations** for those values.



Assignment

Here, we'll work through an example of calculating how old you will be in the year 2040. We'll do this in two steps:

- Subtracting your current age from the current year to get your year of birth.
- Subtracting your year of birth from 2040, to get your age in 2040.

We'll do this in the interactive python shell, which you can access either by running `python3` or `ipython3` for a more nicely colored experience.

Variables

In the steps I described above, there are a few different variables involved, specifically

- Your current age.
- The current year.
- Your year of birth.

In a fresh python session, python doesn't know any of these things. For example, if you run the following code:

```
my_age
```

you'll get the following error:

```
NameError: name 'my_age' is not defined
```

However, we can **assign** your age to the variable `my_age` with the assignment operator, `=`.

```
my_age = 37
```

Running that line of code doesn't result in any output, but now if we re-run `my_age` on a new line, it will print back out whatever numeric **value** you gave it.

Variable Names

There are a few rules about variable names. First, they are case sensitive. We just assigned a numeric value to `my_age`, and if we tried to run `My_age`, or `my_Age`, or anything else that isn't exactly the same as the original variable name, we'll get the `NameError` again. In fact, we can even assign *different* values to variables called `My_age` or `my_Age`, *but I don't recommend it!*

The rules for variable names in python are the following:

- They can start with any upper case or lower case letter, or underscore (in regex `[A-Za-z_]`)
 - They can't start with a number!
- After the first character, they can have any letter, number, or underscore, but that's all.
- No `&` or `*` or `.` or `?` etc. are allowed

Using Variables

Once you've assigned a value to a variable, you can use that variable *like* it was the value. For example, to get the year you were born (± 1), we subtract your age from the current year.

```
2022 - my_age
```

This will subtract the numeric value you assigned to `my_age` as if it was that number. If we assign the value `2022` to the variable `current_year`, we can then use both variables as if they were that number.

```
current_year = 2022  
current_year - my_age
```

We can then assign the output of this subtraction to a new variable called `year_of_birth`.

```
year_of_birth = current_year - my_age
```

If you run `year_of_birth` on its own line, it should print out your year of birth.

To get your age in the year 2040, we just subtract your year of birth from `2040`.

```
2040 - year_of_birth
```

Overwriting variables

Once you've assigned a value to a variable, you can assign a new, different, value to that variable using `=` again. For example, if I wanted to change my age to 50, I could do so like this:

```
my_age = 50
```

Now if we run `my_age`, it will return `50`.

Important!! Changing the value of `my_age` to `50` does **not** update the values of any other variable! The code we ran to assign a value to `year_of_birth` was `current_year - my_age`. Even though we just changed the value in `my_age`, the value in `year_of_birth` has **not** been updated.



Types of Values

There are three basic kinds of values in python

Numbers

Technically, there are two kinds of numbers in Python: integers (numbers without decimal places) and floats (numbers *with* decimal places). This used to be a bigger deal in python2, but python3 converts as necessary. We've already done some work with numbers above. The built in arithmetic in python that we can use on numbers is:

- `x + y` addition
- `x - y` subtraction
- `x * y` multiplication
- `x / y` division
- `x ** y` exponentiation (that is, x^y)
- `x % y` modulus (this gives you the remainder of doing division)
- `x // y` floor division (this gives you the largest whole number that y can go into x)

Strings

In addition to numbers, we also have strings. If we just type text into the prompt, python will think we're asking for a variable, and if such a variable hasn't been created yet, it'll give us a `NameError`. For example, if you type your first name into the prompt, you'll get a `NameError`.

```
Josef
```

However, if you type in the *same* text, but inside of quotation marks this time, it'll just print back out to you the text.

```
"Josef"
```

In this second example, it prints out the text because it's interpreting the text as a **value** instead of a **variable**.

You can put basically any kind of text inside of a string value. There are no restrictions on the characters inside of a string value like there are on variable names. The only complicated thing is when you want to put an actual `"` inside of the string, in which case you need to *escape* it.

```
"I said, \"Hello!\""
```

If we assign a few different strings to variables, we can paste them all together with `+`.


```
title = "Dr."  
first_name = "Josef"  
last_name = "Fruehwald"  
  
title + first_name + last_name
```

Or, we can print them all out nicely, separated by spaces, with `print()`.

```
print(title, first_name, last_name)
```

There is a *lot* more we can do with strings, which we *will* be doing, since most of the linguistic data we're working with comes in the form of text strings!

Boolean, or Logical Values

"Boolean" values are actually named after a guy, [George Boole](https://en.wikipedia.org/wiki/George_Boole)  (https://en.wikipedia.org/wiki/George_Boole). There's only two logical values available, which we access by typing in their name without quotation marks: `True` and `False`.

Usually you won't be typing these values in by hand. Instead, they'll be the result of a *logical comparison*. For example, if I wanted to see if the value in `first_name`, was `"Joseph"`, I could do so with the `==` operator.

```
first_name == "Joseph"
```

That line of code returns `False`, because the value in `first_name` is different.

This is a place to be careful: A common typo or mistake to make is to use the *assignment* operator (`=`) when you mean to use the exactly equal operator (`==`).

While `==` is the only logical operator that works for strings, there are more that work for numbers

- `==` : Exactly equal
- `>` : Greater than
- `>=` : Greater than or equal to
- `<` : Less than
- `<=` : Less than or equal to

Some additional fun operators compare or change logical values. For example, if we run `not True`, we get back the result `False`, and if we run `not False` we get back `True`.

(Note, in the real world, something that is "not false" is not necessarily "true". This is one of many cases where programming might *look* like human language, but behaves differently.)

The two other logical connectives we have are `and` and `or`. The tables below outline what these operators do.

Input	Output
-------	--------

True and True	True
True and False	False
False and False	False

Input	Output
True or True	True
True or False	True
False or False	False

For example, let's say you wanted to check if my first name was "Josef", but you didn't care if it was with an "f" or a "ph". You could check this way.

```
(first_name == "Joseph") or (first_name == "Josef")
```

Or, let's say we wanted to find a "Josef", but only one who is 37. We could do that with:

```
(first_name == "Josef") and (my_age == 37)
```