Data Types

In real life, we classify values as certain types and depending on the type of value, we can do different things to them. The same is done in Python: Python has types of data (some basic, some more complex) and depending on which type of data something is, you can perform certain things to them. Generally, you will be dealing with the four basic data types:

- `str` : read as "string", this data type represents any sort of text enclosed by a pair (or three pairs) of double or single quotes
- `int` : read as int or integer, this data type represents any mathematical integer
- `float` : read as float, this data type represents any number that has a decimal point
- `bool` : read as bool or more formally known as boolean, this data type represents either `True` , `False` , or any value that is equivalent to those two values

I'll refer to these types as strings, integers, floats, and booleans, but I may interchange them with `str` , `int` , `float` , and `bool` . You've used these before (except for `bool`). In Module 1, we used `print` to output strings and in Modules 1 and 2, we used integers to do some basic math. The outputs of some code in Module 3 had numbers that ended with ".0", which means that they are floats.

Strings

Strings are just characters stringed together (bah-duh-tss). A string must have quotation marks on both sides of it to differentiate it from variable names or keywords. These quotation marks must also be the same:

In [ ]:
```
# Example 3-1: Strings!
double_quotes = "Hi!"
single_quotes = 'Hi!'
mixed_quotes = "Hi!'
```

```
  Cell In[1], line 4
    mixed_quotes = "Hi!'
                   ^
SyntaxError: unterminated string literal (detected at line 4)
```

As you can see, the first two variables were made just fine, but the third variable caused an error because I did not enclose the text in the same type of parentheses. This being said, since quotation marks are used to signifiy the start and end of a string, how do we use them in text?

Escape and Special Characters

Well, this problem is solved by attaching a `\` to the start of the quotation mark. `\` undos the special meaning of a character and when used with certain characters, you actually create a new special character. If you wanted to use `\` as text, then you just... add another one in front of it:

In [ ]:
```
# Example 3-2: Escape Characters
sentence = "\"Hello!\", said the dog."
question = 'What\'s your name?'
weird_date = "9\\10\\2023"

print(sentence)
print(question)
print(weird_date)
```

```
"Hello!", said the dog.
What's your name?
9\10\2023
```

As you can see, characters such as `"` , `'` , and `\` were outputted onto the screen although individually, they serve special functions. There are other special characters, notably in text (you'll see these a lot in text files):

- Newline: `\n`
- Carriage Return: `\r`
- Tab (spacing): `\t`

Newline is very useful in forcing the output to continue onto the next line. Tab spacing will be taught in the future and carriage return is something you don't quite need to know in ENGR 102, but you will have to acknowledge its existence.

In [ ]:
```
# Example 3-3: Newline
print("Howdy,\nWorld!")
```

```
Howdy,
World!
```

The (Painful) Game of What Type is the Output?

Python has some quirks when it comes to guessing the type of the output after certain operations. Usually types of hard-coded values are very easy to identify (note: you can use `type()` to check the type of a value!):

In [ ]:
```
# Example 3-4: Basic types
print(type("Hi!"))
print(type(1))
print(type(1.0))
print(type(True))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

Don't worry on why it says `class`: this distinction is irrelevant for this course and all you need to understand is that the desired type was outputted as the second word. But, what if you're given some calculations and you're asked what type gets outputted? This is often troublesome when dealing with numbers because it's not super straightforward. Generally, the rules go like this:

- If you add, subtract, multiply, modulo, raise to a power, or floor divide with two integers, you get an integer
- If you divide any two numbers, you get a float
- If you add, subtract, multiply, modulo, raise to a power, or floor divide with two numbers where at least one of them is a float, you get a float

You could shorten this list to:

- If you use only integers, you only get integers except:
  - If the operation only returns a float (such as division)
- If you ever use a float, the output is a float except:
  - If the operation only returns an integer (very very few exceptions)

In [ ]:
```
# Example 3-5: Outputs Galore!
print(type(1 + 1))
print(type(1 - 1))
print(type(1 * 1))
print(type(1 / 1))
print(type(1 ** 1))
print(type(1 // 1))
print(type(1 % 1))

print()

print(type(1.0 + 1))
print(type(1.0 - 1))
print(type(1.0 * 1))
print(type(1.0 / 1))
print(type(1.0 ** 1))
print(type(1.0 // 1))
print(type(1.0 % 1))
```

```
<class 'int'>
<class 'int'>
<class 'int'>
<class 'float'>
<class 'int'>
<class 'int'>
<class 'int'>

<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

String Concatenation

This is the formal term for adding strings together. When you add two strings together, you take the second string and attach it to the first (simply speaking). Technically speaking, we make a new string and combine the first and second together in the order they are added from left to right. As long as they are strings, they will be combined together as text, regardless if the text can be represented as a number:

In [ ]:
```
# Example 3-6: String Concatenation
print("Hi, " + "my name is" + " Anthony!")
```

```
Hi, my name is Anthony!
```

Note that I manually had to space everything apart. If you choose not to space the strings from one another, they get combined:

In [ ]:
```
# Example 3-6: String Concatenation, No Spaces
print("Hi," + "my name is" + "Anthony!")
print("1" + "2")
```

```
Hi,my name isAnthony!
12
```

Another thing to note is that you can only add a string to another string. If you try adding a number to a string, you will get an error:

In [ ]:
```
# Example 3-7: Failing to add a string and a number together
print("Hi" + 2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 2
      1 # Example 3-7: Failing to add a string and a number together
----> 2 print("Hi" + 2)

TypeError: can only concatenate str (not "int") to str
```

However, you can multiply a string by a number! If you do this, the string gets added to itself as many times equal to the number used (think of it this way: multiplication is just repeated addition) The order doesn't matter as long one value is a number, and the other value is a string. Out of the basic math operators, you can only add two strings together and multiply a string and a number: if you try to use any of the others, you will get errors.

In [ ]:
```
# Example 3-8: String multiplication
print("Hi" * 5)

print("Hi" + "Hi" + "Hi" + "Hi"+ "Hi")

print(5 * "Hi")
```

```
HiHiHiHiHi
HiHiHiHiHi
HiHiHiHiHi
```

Type Conversion

Sometimes, you need your value as a certain value. Usually you do this because a certain function or operation returns a value as one type, but you needed the value as another. There are a bunch of rules to remember when converting between the four primitive types, and they are the following:

- You can only convert a string into an integer if the contents inside represent an actual number (basically if you remove the quotes, is it an integer?)
- You can only convert a string into a float if the contents inside represent an actual number (basically if you remove the quotes, is it any number?)
- If you convert an integer into a float, you get the same number but with `.0` at the end
- If you convert a float into an integer, you only get the whole part (the entire decimal part gets discarded, no rounding occurs)
- If you convert any value into a boolean:
  - If the value is a string:
    - If the string is empty (`""`), then you get `False`
    - Any other string gets you `True`
  - If the value is a number:
    - If the number is equal to 0, then you get `False`
    - Any other number gets you `True`
- If you convert `True` (the boolean) into a...:
  - ...string, you get `"True"`
  - ...integer, you get 1
  - ...float, you get 1.0
- If you convert `False` (the boolean) into a...:
  - ...string, you get `"False"`
  - ...integer, you get 0
  - ...float, you get 0.0

Though a lot of rules, you will soon memorize them pretty quickly. For strings to numbers, the string must clearly just be a number but put inside of quotes (for int, the number specifically has to be a valid integer). Between numbers, you either add the decimal point (integer to float) or remove it entirely (float to integer). There's not really a shorthand for the boolean rules, you'll just have to memorize those although you really won't run into them much if at all.

To actually do the conversion, just use:

- `str(value)` : Converts `value` to a string
- `int(value)` : Converts `value` to an integer

- `float(value)`: Converts `value` to a float
- `bool(value)`: Converts `value` to a boolean

In [ ]:
```
# Example 3-9: Type Conversions
print(int("1"))
print(float("1.0"))
print()

print(float(1))
print(int(1.1))
print()

print(bool(""))
print(bool("hi!"))
print(bool(0))
print(bool(0.0))
print(bool(1))
print(bool(2.1))
print()

print(str(True))
print(str(False))
print(int(True))
print(int(False))
print(float(True))
print(float(False))
```

```
1
1.0

1.0
1

False
True
False
False
True
True

True
False
1
0
1.0
0.0
```

Fancier printing

`print` can actually support multiple values: it's how you enter in these values that determines the formatting and result of the output. Generally speaking, you list multiple values if you are trying to:

- Print multiple items on the same line together (automatically separates with a space)
- Edit the ending
- Edit the separator (for bullet one)

For advanced programmers: This is something I like to explain to advanced programmers a little differently since I really love figuring out the underlying code/patterns/behaviors of well... stuff. If you are unaware of Python's `*args`, Java's varargs (passing an array without a set size as an argument), or anything equivalent, I would skip past this section.

Very basically speaking, the `print` function takes three arguments:

- A list of objects (all objects have a string representation with `__repr__()`): `*args`
- An optional separator to separate each object should there be more than one (defaults to `" "`): `sep`
- An optional ending to end the string with (defaults to `"\n"`): `end`

Thus using `print` is just like calling any other function: call `print` and pass in the necessary arguments to ensure the formatting is proper. Feel free to refer to the code below for an equivalent enough implementation of `print`. Personally I learn better through seeing code, so maybe this can help for more well-versed programmers.

In [ ]:
```
# Advanced Example 3-10: Printing from scratch
import sys

def basic_print(*args: type[object], sep: str=" ", end: str="\n") -> None:
    items = [str(arg) for arg in args]
    new_string = sep.join(items) + end
    sys.stdout.write(new_string)
```

```
basic_print("Hi!", 2, 3.0, True, sep=" | ", end="[:D]\n")
```

```
Hi! | 2 | 3.0 | True[:D]
```

Printing Multiple Things

Instead of adding all the strings together or perhaps converting some values to string and then adding them, you can literally list them in the `print` statement and have them outputted with a space in between them:

In [ ]:
```
# Example 3-11: Printing multiple things
print("Hi! My age is", 20, "and my GPA is", -4.0)
```

```
Hi! My age is 20 and my GPA is -4.0
```

Just separate the values with commas and there you go! Note though this auto-converts values to strings for you, you must be aware of what the string representation of each value is. Some more complex data types have very, very weird string representations and you will want to manually manipulate that value somehow to obtain the desired representation. If you don't want to spaces in between, just specify the separator with `sep=`:

In [ ]:
```
# Example 3-12: Separating things my way
print("Hi! My age is", 20, "and my GPA is", -4.0, sep=" | ")
```

You can also specify the ending to be different (it defaults to newline, which is why usually every `print` statement makes the next line of output start on the next line) with `end=`:

In [ ]:
```
# Example 3-12: Ending things my way
print("Hi! My age is", 20, "and my GPA is", -4.0, end=" || ")
print("It's Miss Rev!")
```

```
Hi! My age is 20 and my GPA is -4.0 || It's Miss Rev!
```

See, no newline this time and instead, the string I used for `end` is used to end the line. You can also use both `sep=` and `end=`. Do note that they must be the last items in the list (you will get an error if you use `end=` or `sep=` in the middle of your list of values). A valid example of using both is shown below:

In [ ]:
```
# Example 3-13: Separatng and ending things my way
print("Hi! My age is", 20, "and my GPA is", -4.0, sep=" /\\ ", end=" || ")
print("It's Miss Rev!")
```

```
Hi! My age is /\ 20 /\ and my GPA is /\ -4.0 || It's Miss Rev!
```

F-Strings

Although you can list multiple values to join them as shown above, there's actually a more user-friendly way to format strings in general: f-strings. F-strings, or formatted string literals, allow you to put in variable values inside the string and format them accordingly. The basic usage of f-string is as follows:

In [ ]:
```
# Example 3-14: F-string basic usage
name = "Anthony"
print(f"My name is {name}")
```

```
My name is Anthony
```

Note two key features:

- The string starts with `f`: this defines the string to be a f-string
- The variable is inserted into a pair of curly brackets

It is within these curly brackets where you insert in a variable or some sort of value and then format that value so that it is displayed in a manner of your choosing. Generally, you want to use f-strings for one of two reasons (at least in this class):

- Alignment + spacing (these two tend to go hand-in-hand)
- Decimal precision

Any formatting specifiers must be applied in this format: `{value:formatting stuff here}`. So within the curly brackets, the value being placed inside the string and formatted must be on the left side of the colon, and all the formatting specifiers must be on the right hand side of the colon.

Alignment + Spacing

You can choose how much space a value should have in the string and whether the value should be left-aligned, right-aligned, or centered:

- Left-aligned (`<`)
- Right-aligned (`>`)

- Centered (`^`)

Right after the colon, use one of those three to set the alignment and then insert a number to specify the width (how much spaces the value takes). Note that if you put a string inside a f-string, you must use quotes that are not the same as the original quotes (in other words, use single quotes if the f-string uses double quotes and vice versa):

In [ ]:
```
# Example 3-15: Alignment + spacing
print(f"|{'Hi!':<10}|")
print(f"|{'Hi!':>10}|")
print(f"|{'Hi!':^10}|")
print(f"|{'Hi!':<1}|")
```

```
|Hi!       |
|       Hi!|
|   Hi!    |
|Hi!|
```

You can see that the alignment is done correctly (note that centering text prefers slightly to the left if the text cannot be perfectly centered), and that the appropriate amount of space is given. If you declared too little space, then the value will just take as much space as needed but no extra (this is as if you printed the value without alignment or spacing). You can also skip the alignment specifier and just do the number, which will adjust how much spacing is needed (and will default to left-align):

In [ ]:
```
# Example 3-16: Just spacing
print(f"|{'Hi!':10}|")
```

```
|Hi!       |
```

Decimal Precision

Number formatting is relatively straight-forward, here's the format: `{value:x.yf}`. `value` is of course, the value we want to put in and format (and in this case, has to be a number). `x` is the same as the spacing as mentioned above, except this time, it's number of digts (and defaults to right-align). As said before, if the number has more digits than `x`, then the number will take up just as much space as possible without taking extra space. `y` is how many decimal points to round to (this is proper rounding so will round down if < 0.5, round up otherwise). `f` is not actually a value but rather the letter: this signifies that this is float formatting, so it will output the decimal points.

In [ ]:
```
# Example 3-17: Decimal Precision
print(f"{3.1415:.2f}")
print(f"{100/3:10.5f}")
print(f"{100/3:<10.5f}")
```

```
3.14
  33.33333
33.33333
```