REPL

The python interpreter, when run in interpreter mode, yields a *read-evaluate-print* loop known as a REPL (pronounced REP-UL).

Numbers

Here is some python code involving integers (type: int) and real numbers (type: float). You can add (+), subtract (−), multiply (∗), divide (/), integer-divide (//), exponentiate (∗∗), and calculate the remainder (%).

- Python follows the PEMDAS order of operations, so order and groupings matter!

```
>>> 1 + 1
2
>>> 2 − 1
1
>>> 3 ∗ 2
6
>>> 4 / 3
1.3333333333333333
>>> 4 / 3 + 1
2.333333333333333
>>> 4 / (3 + 1)
1.0
>>> 4 // 3
1
>>> 8 // 3
2
>>> 3 / 4
0.75
>>> 3 // 4
0
>>> 2∗∗4
16
>>> 5 % 4
1
>>> 9 % 5
4
>>> type(20)
<class 'int'>
>>> type(0.75)
<class 'float'>
>>> type(1 + 1)
<class 'int'>
>>> type(1.0 + 1)
<class 'float'>
>>> int(7.5)
7
```

Every object in Python has a type. Two numeric types are `int` and `float`, and numbers can be converted to either type the builtin functions `int()` or `float()`.

Variables

In python, a variable is a name that refers to a value. Descriptive variable names are a useful way to *document* our code, so it is best to choose names that hint at a variable's purpose. This is part of good programing *style*. There are also some requirements imposed by the language. Variables:

- can contain letters (upper or lowercase), numbers, and _'s
- cannot *start* with a number
- cannot be a Python reserved *keyword*

Here is some code that uses variables (assigns names to values), sometimes correctly, sometimes not. Whenever we make an error in the interpreter, it outputs a message that describes the nature of our mistake. A *syntax error* occurs when we provide code that is not valid according to the rules of the language. In otherwords, our code is malformed. (We will see other types of errors in the future.)

```
>>> my_name = "Bill"
>>> 50cent = "Curtis Jackson"
  File "<stdin>", line 1
    50cent = "Curtis Jackson"
        ^
SyntaxError: invalid syntax
>>> fifty_cent = "in the club"
>>> class = "135"
  File "<stdin>", line 1
    class = "135"
          ^
SyntaxError: invalid syntax
>>> cs135 = "class"
>>> cs135
'class'
>>> Falsey = False
>>> Falsey
False
>>> False = Falsey
  File "<stdin>", line 1
SyntaxError: can't assign to keyword
>>> false = False
>>> false
False
```

So we see that we can use keywords *in* our variable names, but our variable name cannot be *just* a keyword. We also see that case matters. Some keywords have a value, like `True` and `False`, and can therefore be assigned to a variable. Others, like `class` have a specific meaning in the language, but not a value.

Strings

A Python *string literal* can be formed by enclosing text in a pair of apostrophes like 'this' or a pair of quotes like "this". Unlike parentheses, which have a notion of a "left" and a "right", these symbols do not. So we must construct strings in a way that has no ambiguity. If we want to use only apostrophes in our text, we can enclose our text in quotes. If we want to use only quotes, we can enclose our text in apostrophes. If we want a mix, we must use *escaping*, which is done with the backslash (\).

We can also use the escape character to create other special meanings, including a newline (`\n`), a tab (`\t`), or a literal backslash (`\\`). Why do we need to escape a \?

```
>>> x = "Brent's sister's husband's brother-in-law is a great guy."
>>> x
"Brent's sister's husband's brother-in-law is a great guy."
>>> y = 'Brent says, "Good thing my brother-in-law is an only child."'
>>> y
'Brent says, "Good thing my brother-in-law is an only child."'
>>> z = x + " " + y
>>> z
'Brent\'s sister\'s husband\'s brother-in-law is a great guy. Brent says, "Good thing my brother-in-law is an only child."'
>>> print(x)
Brent's sister's husband's brother-in-law is a great guy.
>>> print(y)
Brent says, "Good thing my brother-in-law is an only child."
>>> print(z)
Brent's sister's husband's brother-in-law is a great guy. Brent says, "Good thing my brother-in-law is an only child."
>>> a = "a newline\ncharacter"
>>> print(a)
a newline
character
>>> a = r'a newline\ncharacter'
>>> a
'a newline\\ncharacter'
>>> print(a)
a newline\ncharacter
```

Consider the following interaction on the Python interpreter. What is `x`?

```
>>> print(x)
She said, "Brent's favorite character is \n."
He said, "I know."
```

Python

Let's write a program called `sum.py` that takes two arguments from the command line and prints out their sum.

```
1 import sys
2
3 x = sys.argv[1]
4 y = sys.argv[2]
5
6 print("The sum of " + x + " and " + y + " is " + (x+y))
```

First some explanation. The `import` command tells Python to include a bundle of code, called a module, in our program. Importing a module gives us access the variables and functions it defines. The module `sys` gives us access to a variable called `argv`, which is a vector of strings that appear on the command line. `sys.argv[0]` is the name of the script. `sys.argv[1]` is the first argument, `sys.argv[2]` is the second argument, and so on. Let's run this script in script mode.

```
1 $ python3 sum.py 5 6
2 The sum of 5 and 6 is 56
```

Um, that's not right. What's wrong? The arguments are strings of characters, not numbers.

```
1 import sys
2
3 x = int(sys.argv[1])
4 y = int(sys.argv[2])
5
6 print("The sum of " + str(x) + " and " + str(y) + " is " + str(x+y))
```

In the code above, we convert a valid string into an integer using `int()`, and we convert that interger back into a string using `str()`. What would happen if we didn't convert `x` and `y` back to strings in the `print()` command? What would happen if we passed arguments to our script that were not integers?