

The Python interpreter

You can run the Python interpreter to run your code by just typing `python3` (or `py` on Windows) in a Terminal window. It takes over the Terminal and you should see `>>>` to the left, which means it's ready to interpret your Python commands.

The interpreter is useful for **debugging**, where you run pieces of code from a larger program you are writing to figure out why it's not doing what you intend. We did a little debugging on Wednesday in Playgrounds.

Operators and precedence

Basic binary and unary operators

The Python interpreter is useful to run small numbers of commands to see what they do and check your understanding, as we will often do in class. For example, the interpreter can do basic arithmetic:

```
>>> 2+2
4
```

```
>>> 2**4
16
```

```
>>> 1024/8
128
```

Common operators that Python supports include `+`, `-`, `/`, `*`, `**` and `abs()` for absolute value:

```
>>> abs(-10)
10
```

Also there's the **remainder operator (modulus)** `%` (remember long division):

```
>>> 13 % 3
1
```

And the **floor division operator** `//` if you just want to know how many whole number times does a number divide into another:

```
>>> 13 // 3
4
```

So between the remainder and the floor division operators, the results should combine to give you the original integer you divided.

Precedence rules

Like in algebra (remember **PEMDAS**?), if you use multiple operators in a single expression, there are **precedence rules** to determine which order of operations. ****** takes the highest precedence, followed by negation **-x**, followed by *****, **%**, and **/**, followed by **+** and **-**. When in doubt, use parentheses, which always take the highest precedence:

```
>>> -1**2
-1
>>> (-1)**2
1
```

math module

What if you want to do more complex math calculations? The Python interpreter makes for a *great* scientific calculator if you run the command `from math import *`.

Python comes with many **modules** (also called **packages** or **libraries**) that contain useful **functions** (in the sense that we learned about last time):

```
>>> exp(log(3))
20.085536923187668

>>> 100*log(e)
100.0

>>> sqrt(cos(pi)**2 + sin(pi)**2)
1.0
```

Here, `math` is the module name (thing that contains functions like `sqrt`, `exp`, `log`, etc.), the `import math` piece of the command tells Python to load the module named 'math', and the full statement `from math import *` means load all the functions from 'math' into memory (`*` is a wildcard, meaning 'all' in English).

A **statement** means a valid, full line of code, somewhat analogous to a 'complete sentence' in English (e.g. `from math` would be an 'incomplete sentence' in Python and generate an error).

You can read about all the functions online in the [math module online](#).

In lab and in your project, you import functions from the module `turtle`: `from turtle import *`.

random module

You can add some randomness to your programs with the `random` module. It can be imported just like `math`: `from random import *`. There are lots of useful functions here, but I'll introduce you to two. First, `random()` generates a random number between 0 and 1. It includes 0 in that range, but not 1, which I'll write [0, 1).

```
>>> random()  
0.7660076867816247
```

How could we generate a random number between 0 and 15?

How about between 5 and 20?

Second, if you only want whole numbers between a and b inclusive [a, b], `randint(a, b)` does the job:

```
>>> randint(3, 6)  
5
```

You can read more about this module and what functions are available in the online Python documentation.

Data types

Now that we know basic kinds of operations we can perform in Python and some useful modules, what kinds of data can it process?

If you were going to design a system that processes data, what fundamental types of data would you want to make sure it incorporated and accounted for?

Python has four basic types of data (**data types**): integers (**ints**), numbers with values after the decimal point (**floats**), text (**strings**), and truth values (**Boolean**).

Here are examples of each:

```
>>> 2+2  
4
```

```
>>> 19/2  
9.5
```

```
>>> 'Hi CS151!'  
'Hi CS151!'
```

```
>>> True  
True
```

Notes:

1. Here, each `2` is treated as a numerical **integer (int)**. Notice that unlike the string example, Python **evaluates** the operation `2+2` to get 4, then prints the result to the console. Other examples of ints include -101, 2018, 0, 5.

2. Python evaluates `19/2` and prints the result to the console, which is a **floating point number** (or **float** for short). You can tell because unlike ints, floats have a decimal point in the number. Other examples include `10.0`, `3.14159265`, `0.01`
3. Here we display the literal text `'Hi CS151!'`. The single quotes tell Python to treat the enclosed words as a literal, textual phrase and not try to treat it as a command (like `2+2` in the next example).
 - This kind of 'text' data, enclosed by either single `"` or double quotes `"` (your choice, but pick one and stick with it), is called a **string**.
4. **Boolean**: Either `True` or `False` (note that the uppercase first letter is important, Python won't understand `true`).

Checking data types

If you're in doubt about what the data type is, you can use Python's `type` function:

```
>>> type(32.0)
<class 'float'>
>>> type(False)
<class 'bool'>
```

Printing information to the console

What if we wanted to see the output of some math not in the interpreter but in a Python code file (`.py` file)?

Typing the math out and running the program will not show us the result!

```
2 + 2 # Nothing shows up if placed in code file
```

We need to use the `print` function to display information to us in the Terminal when running code in a Python file. As you write code in Project 1 and beyond, the `print` function will

1. be useful to display information about what's going on inside your program
2. show the result of problems you solve
3. provide instructions to users about how to use your program.

Ok here are examples of using `print` to display information:

```
print(2+2) # 4 in Terminal
print('Hi CS151!') # Hi CS151! in Terminal
```

Like with the `2+2` example, in the case of a string **print evaluates it's input inside the parenthesis**. Print shows us the text without the quotes, because the text is the actual data, not the quotes.

Printing multiple data at once

There are two ways to print multiple pieces of data from your problem.

Multiple print statements

Each time that you call print in your programs, it creates a new line, so if you call print multiple times in a row, the text gets printed on separate lines:

```
print(1)
print(2)
print(3)
```

...results in:

```
1
2
3
```

You can also call print without an argument and it will print a blank line to the console:

```
print(1)
print()
print(2)
```

...results in:

```
1

2
```

Print with multiple arguments

You can print multiple things out at once with a single `print()` statement because `print()` accepts multiple **arguments**. The syntax is `print(thingA, thingB, ..., thingN)` and arguments can have different data types:

```
>>> print('The result of my program is', 90/15, ' and there are ', 1+1+1+1+1, ' arguments in this
The result of my program is 6.0 and there are 5 arguments in this print statement.
```

Example: Debugging with turtle

Print statements can be very useful when debugging your labs and projects. For example, `turtle` has functions that tell you where turtle currently is (`position()`) and what direction it's facing (`heading()`). You can use these in print statements to help you understand the turtle's current state when it does something you don't expect:

```
# perform a sequence of steps
forward(50)
right(33)
backward(77)
left(42)
forward(11)

# print out info about the turtle's state
print('The turtle is positioned at (x,y) =', position(), 'and is facing', heading(), '.')
print('The distance to the origin is', distance(0, 0), '.')
The turtle is positioned at (x,y) = (-3.71,43.66) and is facing -9.000000000000002
```

Operator data type compatibility

How Python represents or stores your data matters! You can only perform certain operations on different data types, and the results might not make sense if you're not careful.

Concatenation and replication

For example, `'405' + '1'` results in `'4051'` because they are both strings, so Python **concatenates** them (i.e. 'glues the strings together'). As we just said, putting the `+` operator inside a string won't perform addition because strings are not evaluated a numeric data (e.g. `'405 + 1'` is `'405 + 1'`). Also `'405' + 1` results in an error because it doesn't make sense to add a string and an int.

Multiplying a string and a int *does* work (**replication**):

```
>>> print('Hi There!' * 10)
Hi There! Hi There! Hi There! Hi There! Hi There! Hi There! Hi There! Hi There! Hi There!

>>> print('456' * 3)
456456456
```

But on the other hand, multiplying two strings together does NOT work!

```
>>> 'My name is' * 'Byte'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Be careful with data types: multiplying the a string and an int gives you repeating text, while multiplying a string with the same number represented as a string gives you an error.

Combining floats and ints

- floats mixed with integer division (/). e.g.

```
>>> 10.0 // 3 # 3.0
```
- To clarify: / is **floating division**.

```
>>> 10 / 2 # 5.0
```
- // is **integer division**

```
>>> 10 // 2 # 5
```
- **Rule:** Python makes the result of any math operation that involves a float a float.

Type conversion

Example: Comparing an int and a float

What's the resulting type below?

```
>>> 4.0 ** 3
```

How does Python arrive at `64.0` ? Python has to make sense of how to combine a float and an int. One option is to convert everything into ints, then take the third power.

Converting may result in information loss

Converting floats to ints may result in information loss. You can see this if we force the following float into an int:

```
>>> int(3.9)
3
```

`3.9 ** 3` would be very different than `3 ** 3` ! So Python converts each type to float then exponentiates.

You can convert types yourself, like I did above, using `int()` to convert to an int, `float()` to convert to a float, `str()` to convert to a string, and `bool()` to convert to a Boolean.

If you'd rather round to the nearest integer when converting to ints, rather than simply discarding the decimal part (called taking the **floor**; there's also a ceiling function **ceil** to always round up), you can use the Python function `round()` :

```
>>> round(3.9)
4
```