CSSE 120 – Introduction to Software Development

Getting started in Python:

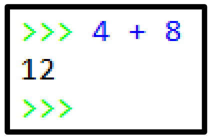
Open a Python Console as described in a previous document. Throughout these exercises, type within your Python Console to "***evaluate***" things as directed, which means:

- Type the item at the prompt in the Python Console.
- Press *Enter.*
- Examine what the computer spits back, and
- ***Make sure you understand why that expression yields that result***.

For example, the first exercise below says to evaluate:

```
4  + 8
```

So, you type that in the Python Console (after the **>>>** prompt). You will see that the computer spits back **12**. (Duh!) Assuming that you see ***why*** it spit back **12**, you continue to the next step of the exercise.



```
>>> 4 + 8
12
>>>
```

As you work through these exercises, **do the associated Moodle quiz.** ***Answer the questions in the Moodle quiz WITHOUT typing those questions into the Python Console,*** as best you can; that way, you can self-assess how well you are understanding the concepts in this exercise.

- Don't just type – make sure you ***understand*** what is happening.
- **BRING YOUR QUESTIONS TO CLASS.**

You will find many of these exercise straightforward. But you may find some of them quite mysterious. **That's OK – just *ask questions* in class until you understand the concepts shown in these exercises.**

Part 1: Numbers, Arithmetic, and Precedence

1. You can do the **usual arithmetic operations** with numbers. Try evaluating the following in the Python Console:

   ```
   4 + 8
   7 * 10
   1.53 + 8
   ```

2. Do some ***subtraction*** and ***division*** using examples that you choose.

   Yes, do it now in your Python Console!

3. We call things like `+` ***operators***. It is a ***binary operator*** because it needs TWO things, one on its left and one on its right. (Binary means "two".) We call things like `7 * 10` and `4 + 8` and `(4 + 2) * 3` ***expressions***.

   You can use parentheses to make sense of expressions with more than one operator. Try these:

   ```
   (4 + 2) * 3
   4 + (2 * 3)
   4 + 2 * 3
   ```

   Parentheses matter! Use them, especially at first, so that you don't have to remember all the so-called ***precedence*** rules.

4. The ***exponentiation*** (raising to a power) operator is the `**` symbol. Try it:

   ```
   2 ** 10
   10 ** 2
   2 ** 0.5
   ```

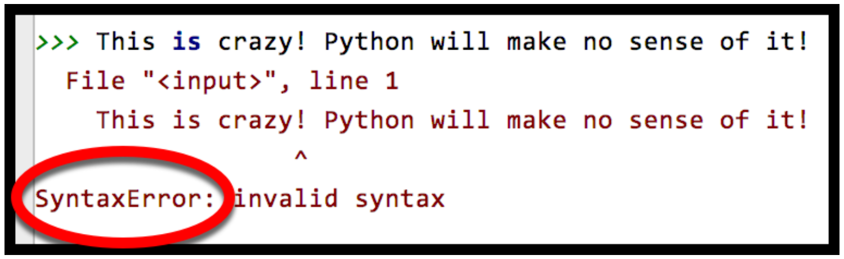
5. Do more examples until you are sure you understand ***exponentiation*** in Python. Yes, do it now!

If you have not already done so, **do Part 1 of the Moodle quiz** (on *Numbers, Arithmetic, and Precedence*).

Part 2: Syntax and Run-Time Errors; Exceptions

6. Python programs have a ***notation***, or ***syntax***, that is required for the Python interpreter to execute (run) the program. Violations of the required notation are called ***syntax errors***.

   PyCharm notes such errors even before you run the program. If you run a program that executes a statement that has a syntax error, you get a run-time message like in this example:



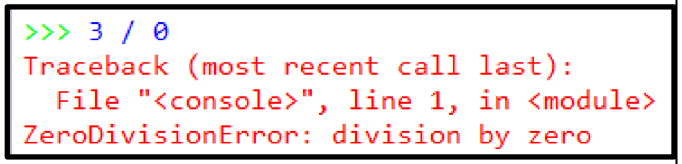
```
>>> This is crazy! Python will make no sense of it!
  File "<input>", line 1
    This is crazy! Python will make no sense of it!
                ^
SyntaxError: invalid syntax
```

   Try typing and then evaluating some short, silly thing that you think might include syntax errors, like the above. (*Reminder*: You should be typing things in your Python Console, as directed in these exercises.)

7. You would also expect bad things to happen from ***bad arithmetic***. Try:

   `3 / 0`

   You should see a red error-message like the one to the right.



```
>>> 3 / 0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: division by zero
```

   Always read the last line of the error message – it summarizes the error. If the message makes no sense to you, ask someone to clarify.

   Errors like these are called ***run-time errors***, aka ***exceptions***.

8. You would expect bad things to happen from other ***bad expressions***. Try:

   `3 / hello`

   and **read the last line of the error message.** If that message makes no sense to you, ask someone to clarify.

9. Write another couple of expressions that cause error messages. Be creative! Be sure you understand the error messages.

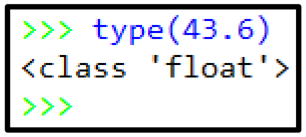
   Yes, do it now!

If you have not already done so, **do Part 2 of the Moodle quiz** (on *Syntax and Run-Time Errors; Exceptions).*

Part 3: Objects, Types, and Values

10. In Python, every "thing" (that is, every item of data) is called an ***object***.

    An ***object*** has a ***type*** and a ***value***. Let's learn some *types*:



```
>>> type(43.6)
<class 'float'>
>>>
```

    The **`type`** function tells you what the type of an object is, as shown in the Python Console snippet above. Types are sometimes called ***classes***; two ways to say the same thing, in this context.

    Use the ***type*** function per the example above to determine the type of each of the following objects.

```
482
48.203
"blah blah blah"
'yada yada'
[4, 2, 9]
min
min(4, 6, 2, 12, 10)
min(4, 6, 2.0)
```

Those are ***double-quote*** characters (*SHIFT-single-quote* on your keyboard), NOT two single quotes in a row!

Do you see why the types of the last three expressions are *different?* If not, ask someone!

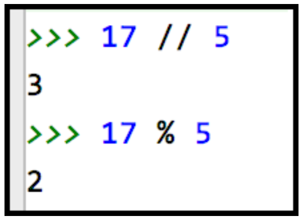
11. Objects of type ***int*** and ***float*** can do the **usual arithmetic operations**:

    - Type an expression that involves addition, subtraction and multiplication (but NOT division, yet), using ***whole numbers*** (which are of type **int**).
    - Repeat the above, but making just a single one of the numbers in your expression a ***float***, by appending a decimal point to it, like this: instead of **2** (which is an *int*) write **2.0** (which is a *float*).
    - Now try division:

```
4.2 / 2.0
4.2 / 2
4 / 2
3 / 2
```

    What do you notice about the type that results from division, even if both arguments are int objects?

12. You can do integer division using the **//** operator and you can get the remainder when you do integer division by using the **%** operator. For example, 5 goes into 17 three whole times with a remainder of 2, per the example to the right.



```
>>> 17 // 5
3
>>> 17 % 5
2
```

    Try out the **//** and **%** operators on some more integers until you understand what they do.

13. Objects of type ***string*** can do a sort of arithmetic. Try expressions like this (feel free to use your own strings and numbers):

```
"hello" + "goodbye girl"
"big" * 20
("hello" + "goodbye girl") * 8
```

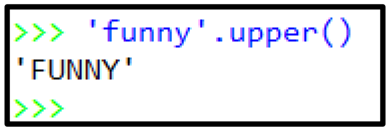
'single quotes' + "double quotes"

14. In Python, both single and double quotes can denote strings. Use ***both*** to figure out how to write a string that has a single-quote as part of the string (for example, the contraction ***isn't***). Ask for help if you don't see how to do this!

    Yes, do it now!

15. The ***type*** of an object determines what an object knows and what it can do. For example, strings know their characters and can return an upper-case version of them, like this:

    'funny'.upper()

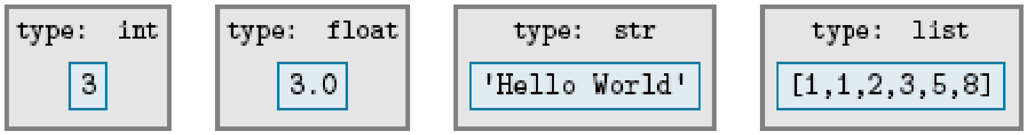


```
>>> 'funny'.upper()
'FUNNY'
>>>
```

    Try it, using your own string!

    We will talk more about this "dot" notation later in these exercises.

To summarize: An ***object*** has a ***type*** and a ***value***. For example:



The ***type*** of an object determines:

- The ***kind of thing*** the object is
- What sort of thing the object ***knows*** (stored in what we call ***instance variables***) and what the object can ***do*** (***operations***, like you have just seen, and more generally what's called ***methods***).

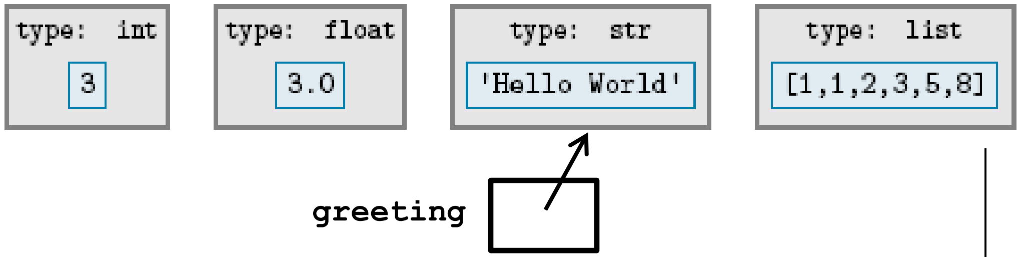
Lots more on this shortly! Don't worry if it does yet make total sense!

If you have not already done so, **do Part 3 of the Moodle quiz** (on *Objects, Types, and Values*).

Part 4: Names, Variables, and Assignment

16. We use ***names*** to ***refer*** to objects. We often use the word "*variable*" instead of "*name*" since the value of the object to which the name refers can *vary* over time.

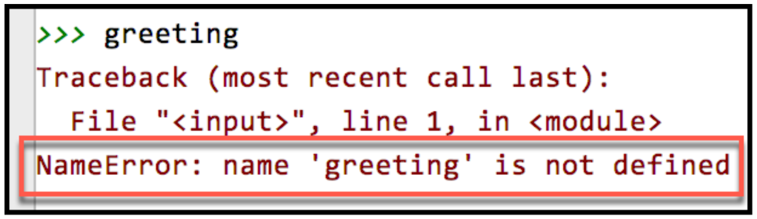
For example, the name ***greeting*** might refer to the object per the diagram below:



The ***value*** of a name is the object to which the name refers. A name gets its value through ***assignment***. One way to do assignment is by using the **=** (read it as "***gets***") operator. For example, to make ***greeting*** refer to the object above, we would write:

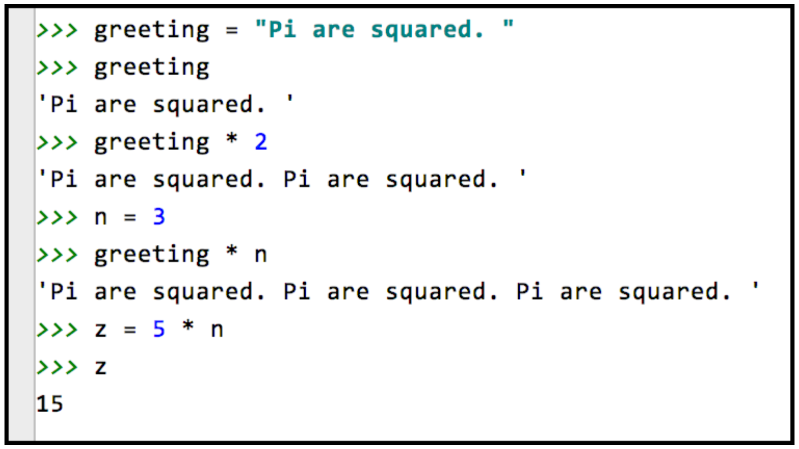
```
greeting = 'Hello World'
```

Until you assign a variable a value, it has no value; we say that it is ***not defined***. For example, if you typed ***greeting*** in a Python Console before typing an assignment statement like the above, you get an error message, as shown below.



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

Once you have assigned a name a value, it evaluates to that value in all subsequent uses in the Python Console. Here is an extended example (in the next column):

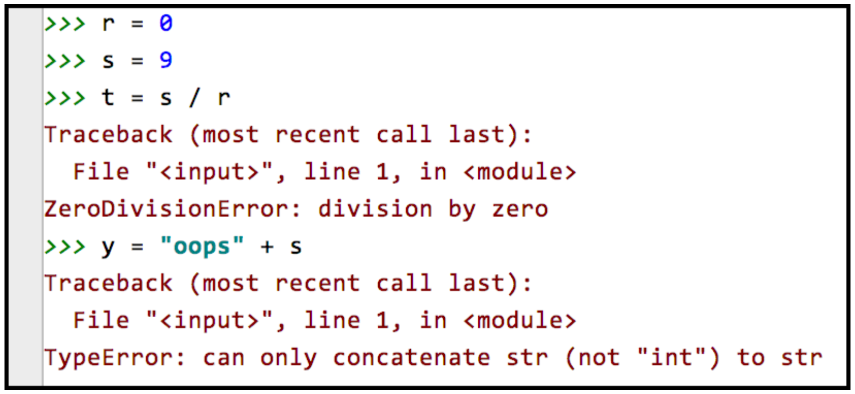


```
>>> greeting = "Pi are squared. "
>>> greeting
'Pi are squared. '
>>> greeting * 2
'Pi are squared. Pi are squared. '
>>> n = 3
>>> greeting * n
'Pi are squared. Pi are squared. Pi are squared. '
>>> z = 5 * n
>>> z
15
```

Practice assignment as suggested by the above example, that is: Choose your own names, given them values by using the assignment (=) operator, and define new names by using expressions that include names that you defined previously.

Yes, do it now!

17. Assignments can result in run-time errors (discussed previously), as in these examples:



```
>>> r = 0
>>> s = 9
>>> t = s / r
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
>>> y = "oops" + s
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

As always, **read the last line of the error message and try to make sense of it.** If the message makes no sense to you, ask someone to clarify. (For example, do you understand the error message in the 2nd example above?)

Try some assignments that yield run-time errors. ***Yes, do it now!***

18. The assignment operator is not symmetric! That is, when an assignment statement `blah = such-and-such` executes:

- ***First,*** the ***right***-hand-side is evaluated (computed), thus yielding an object.
- ***Then,*** the name (variable) on the ***left***-hand-side is made to refer to the object on the right-hand-side.

There is a big difference between the two sides of an assignment! Try the following (some of which yield error messages):

Throughout these examples I will use short, silly names – ***in real programs the names (variables) would have meaning*** and hence you would ***choose meaningful names for them***.

```
a = 45
45 = a
b = 10
c = b + 20
b = c
```

(and then enter `b` *to see what* `b` *evaluates to now, and likewise enter* `c` *to see what* `c` *evaluates to now)*

Make sure that you understand the ***non-symmetric*** aspect of the assignment operator `=` now. Ask questions as needed!

19. Finally, see if you can make sense of this:

```
x = 5
x = x + 1
```

(and then enter `x` *to see what* `x` *evaluates to now)*

If you understand why `x = x + 1` is perfectly sensible computer science (but lousy mathematics), then you understand the assignment operator. ***Ask questions as needed about this KEY IDEA!***

If you have not already done so, **do Part 4 of the Moodle quiz** (on *Names, Variables, and Assignment*).

Part 4: Calling functions

20. One last concept: let's see how you ***call functions***, where a function is just like in mathematics (in this context).

Functions are defined in ***modules*** (Python's word for a file that contains Python code – sometimes we also call a collection of one or more modules a ***library***). Many of the mathematics functions are in the `math` module. The notation for using such functions goes like this:

- First, you ***import*** the relevant module (i.e., library), e.g.:

  ```
  import math
  ```

  Doing that *once* per Python Console run is enough – the above ***import*** statement "sucks in" the definitions of all the math functions, for you to use in the rest of the run.

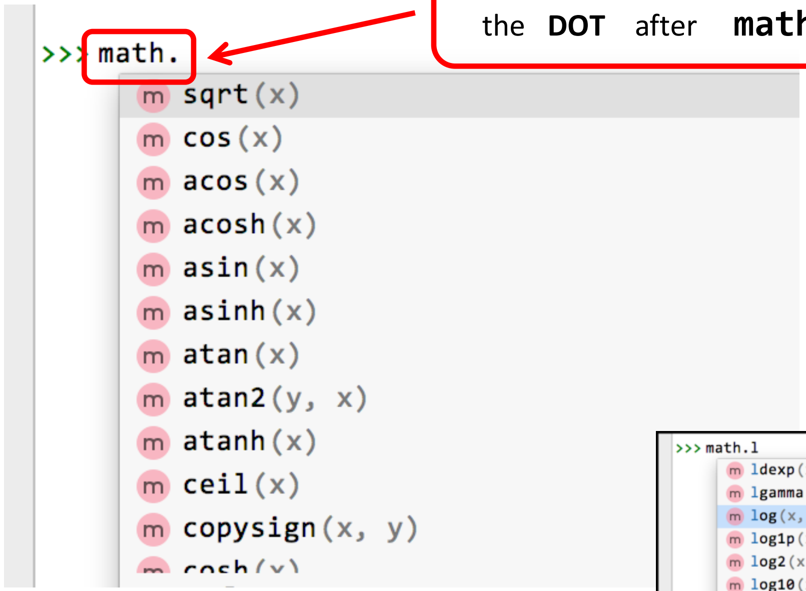
- Then, you precede the function name by the module name and a dot (aka period, full stop), like this:

  ```
  math.sin(3.14)
  x = math.pi
  math.cos(x)
  ```

Try the above (don't forget to do the `import math` first).

Yes, do it now!

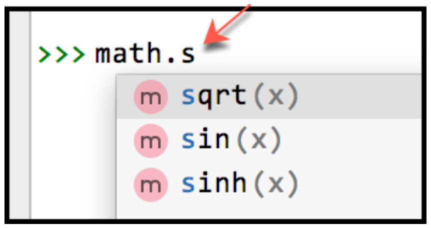
21. PyCharm does a wonderful thing to help you learn the names of the functions in a library – after you type the **DOT** after `math`, if you ***pause*** at that point, ***PyCharm pops up all the functions*** (and other things) ***in the math library,*** as shown on the next page:



```
>>> math.
    sqrt(x)
    cos(x)
    acos(x)
    acosh(x)
    asin(x)
    asinh(x)
    atan(x)
    atan2(y, x)
    atanh(x)
    ceil(x)
    copysign(x, y)
    cosh(x)
```

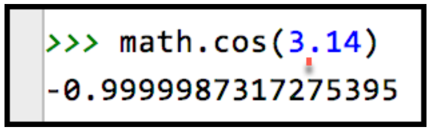
This is what I typed. Note the **DOT** after `math`

You can scroll through the list that appears or you can type one or more characters after pausing at the DOT, like this example that shows me all the math functions that begin with "s":



```
>>> math.s
    sqrt(x)
    sin(x)
    sinh(x)
```

Using this "dot trick", ***try some other functions*** from the `math` library until you feel comfortable with them. For example:

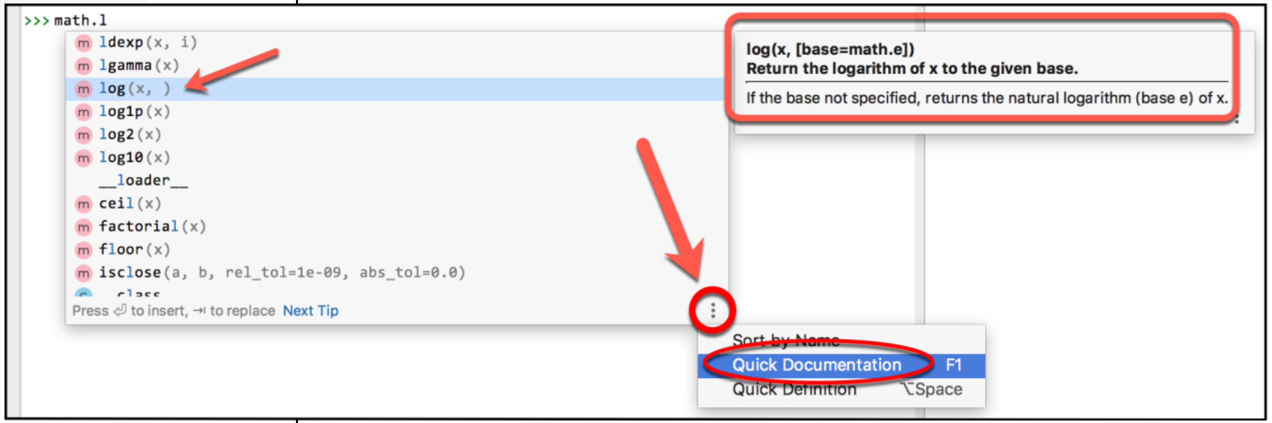


```
>>> math.cos(3.14)
-0.9999987317275395
```

Yes, do it now!

22. Assign a name the value `math.pi` and then compute the cosine of the value of that name. Do the trigonometric functions seem to use radians or degrees?

23. You can even get ***documentation*** on the functions when using the DOT trick, by bringing up the DOT selection (i.e., pausing after typing the DOT) and then (per the picture below):
    - Select the ***three dots*** on the ***bottom-right***, then
    - Select ***Quick Documentation***, then
    - Select the function whose documentation you want.



```
>>> math.l
    ldexp(x, i)
    lgamma(x)
    log(x, )
    log1p(x)
    log2(x)
    log10(x)
    __loader__
    ceil(x)
    factorial(x)
    floor(x)
    isclose(a, b, rel_tol=1e-09, abs_tol=0.0)
Press ⏎ to insert, ⇥ to replace  Next Tip
```

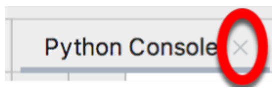
log(x, [base=math.e])
Return the logarithm of x to the given base.
If the base not specified, returns the natural logarithm (base e) of x.

24. The functions (and other things) in the `builtins` module are well, "built in". So you don't have to type the ***builtins-dot*** in front of them (and would not ordinarily do so). Try:

```
import builtins
builtins.abs(-45)
abs(-45)
min(55, 3, 20, 4)
```

Everyone uses the shorter form. But even the ***builtins*** are from a library, just like the ***math*** functions. There are many *thousands* of Python libraries.

If you have not already done so, **do Part 5 of the Moodle quiz** (on *Calling Functions*). ***Then exit your Python Console*** by clicking on the little X beside the words "Python Console" at the top of the Python Console.


Python Console