

CSSE 120 – Introduction to Software Development

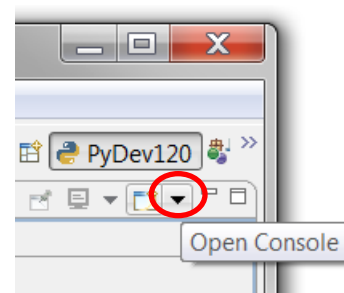
Getting started in Python – Numbers, Arithmetic Operators, Expressions, Objects, Types, Variables, Assignment, Calling Functions (whew! 😊)

Instructions:

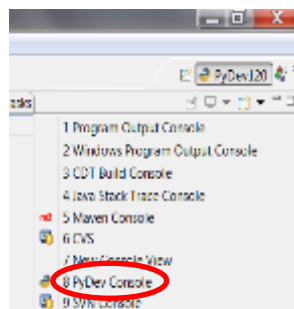
In this exercise, you will type Python commands into a **PyDev Console** or (if the PyDev Console does not work for you) in a program called **IDLE**. Here are instructions for opening a PyDev Console, followed by instructions for running IDLE.

To open a PyDev Console, in Eclipse:

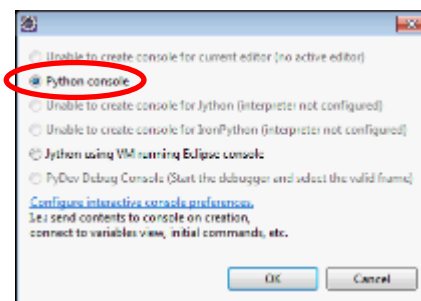
- Start with the pull-down arrow (in the upper right of Eclipse, on the Console tab) that brings up an Open Console pop-up if you hover. See the picture to the right. (It's a little tricky to locate. Ask someone to show you if you can't find it).



- From the pull-down menu, select **PyDev Console**



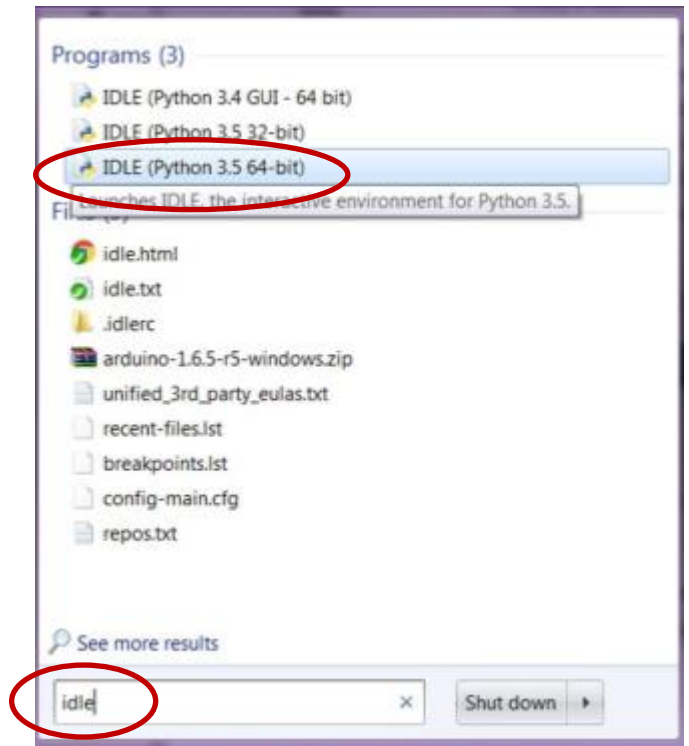
- In the dialog box that pops up, select **Python console**.
- Press **OK**.



IMPORTANT: If you get an error message at this point, use IDLE, as described on the next page. Don't use the Jython option. IDLE is easier and better.

To run IDLE: (Do this if the Pydev Console does not work for you. This is NOT in Eclipse.)

- At the **Start** menu, type **idle**, as shown in the picture to the right.
- Among the **Programs**, you should see at least one labeled **IDLE**. If you have more than one, choose the one that says Python 3.5.
- Something that looks like the picture below should pop up. **It won't look EXACTLY like the picture**, since I am using a Mac. I have typed `2 + 2` in the picture, and IDLE has shown the result, which is `4`. IDLE is waiting for me to type something at the triple-arrows.



```

Python 3.5.1 Shell
Python 3.5.1 (v3.5.1:37a07cee5969, Dec 5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
2 + 2
4
>>>
Ln: 8 Col: 4

```

Continues on the next page.

Instructions (continued): At this point, you should have open either a PyDev Console or IDLE. In either case, you can type in Python expressions.

Work your way all through this document. But note:

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

You will find much of this exercise straightforward. But you will probably find some of it quite mysterious. **That's OK if you ask questions until you understand the concepts shown in this exercise.**

- The instruction “*evaluate the following*” means: Type the item at the prompt in the PyDev console (or IDLE), 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 PyDev 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
>>>
```

Your console may look slightly different than the pictures in this handout.

For example, yours may say **Out[1]: 12** in the above example. No problem.

Numbers and arithmetic. Operators and expressions. Parentheses and precedence.

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

$$4 + 8$$

$$7 * 10$$

$$1.53 + 8$$

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

Yes, do it now!

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 ******
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!

Exceptions, run-time errors.

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

```
3 / 0
```

You should see a red error-message like the one to the right. Read it. If it makes no sense to you, ask someone to clarify.

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

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

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

```
3 / hello
```

You should see a red error-message. Read it. If it makes no sense to you, ask someone to clarify.

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

Yes, do it now!

(This exercise continues on the next page.)

Objects and Types. Int, float, string.

9. 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**, like this:

The **type** function tells you what the type of an object is, as shown in the PyDev Console snippet to the right. Note that types are sometimes called **classes**; two ways to say the same thing, in this context.

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

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!

10. 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**'s?

Numbers can also do other operations that we’ll learn about later.

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

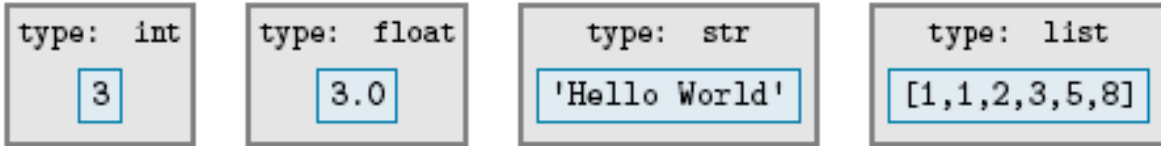
```
'hello' + 'goodbye girl'
'big' * 20
('hello' + 'goodbye girl') * 8
'single quotes' + "double quotes"
```

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

12. 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!

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 kind of thing the object **knows** (we call those **instance variables**) and what kind of thing the object can **do** (**operations**, like you have just seen, and more generally what's called **methods**).

What does an object know and what can an object do?

13. 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!

We will see LOTS more of this

“who DOT does-what PARENTHESIS with-what PARENTHESIS”

notation!

(This exercise continues on the next page.)

Dynamically typed versus statically typed.

15. In Python, variables don't themselves have types. If you ask for the type of a variable, what you are really asking for is the type of the object that the variable refers to. Languages in which variables don't themselves have types (but the objects to which they refer do) are called **dynamically typed** languages – Python is one such language. **Statically typed** languages that you might have heard about include C, C++ and Java.

Try these (one at a time, in this order), and use **type(x)** between each to determine the (current) type of the variable **x**.

```
x = 45
x = -5.32
x = x + 10
x = 'hello' * 20
```

Remember, do **type(x)** between each of these to see the *current* type of the variable.

What do you notice about the type of the variable **x** – does it stay the same throughout or change?

Also, after you have changed **x** from **45** to something else, can you “get back” the **45**?
(Answer: No, not in this context.)

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

16. 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 the latter is perfectly sensible computer science (but lousy mathematics), then you understand the assignment operator. **Ask questions as needed about this KEY IDEA!**

The assignment operator is not symmetric.

17. When an assignment statement **blah = such-and-such** executes:

- **First**, the **right**-hand-side is evaluated (computed), thus yielding an object.
- **Then**, the variable (name) 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):

```
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!

Calling functions.

18. Next, 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 need to **import** the relevant module (i.e., library).

```
import math
```

You need to do that only *once* per PyDev run – it “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, like this:

```
math.sin(3.14)
```

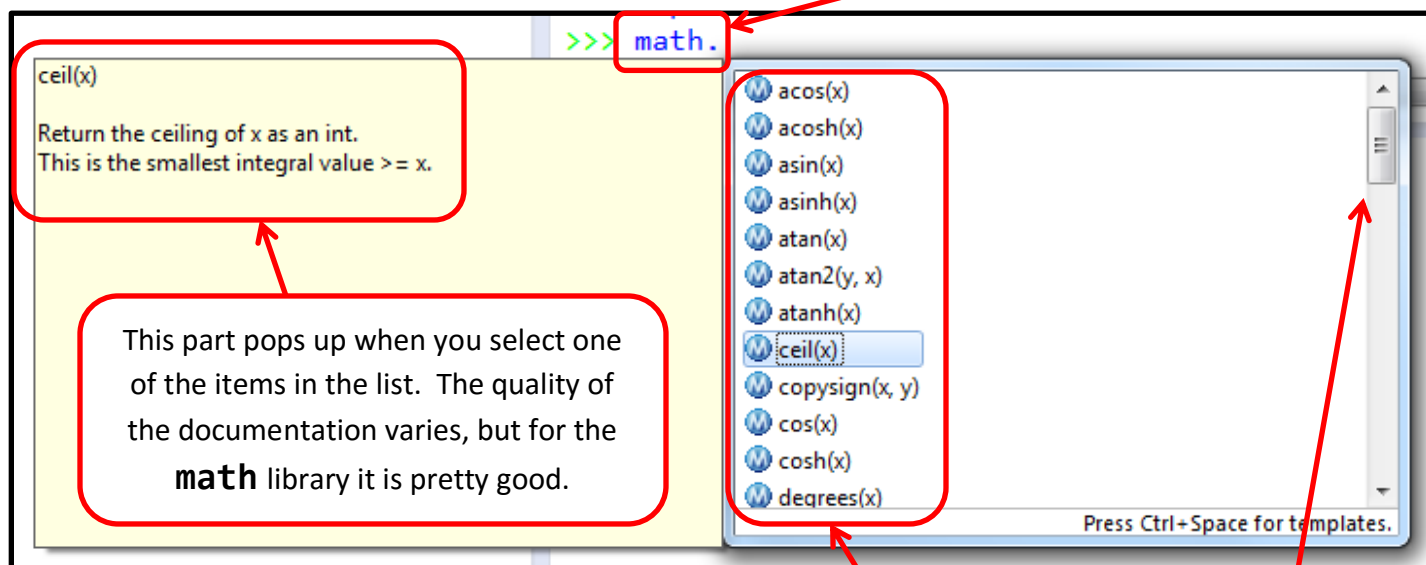
```
x = math.pi
```

```
math.cos(x)
```

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

Yes, do it now!

19. PyDev 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, **PyDev pops up all the functions** (and other things) **in the math library**, like this:



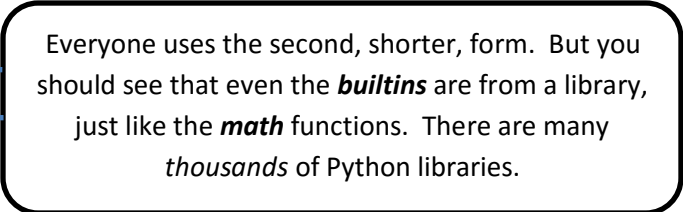
Using this “dot trick”, **try some other functions** from the **math** library until you feel comfortable with them.

Yes, do it now!

This part pops up first. You can *scroll* to see ALL the functions (and other things) in the **math** library!

20. 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 second, shorter, form. But you should see that even the ***builtins*** are from a library, just like the ***math*** functions. There are many *thousands* of Python libraries.