labs. Please make use of these resources.

Topical Syllabus

Why a "topical" syllabus?

Each group of students is unique and so, in different classes, we need to spend more or less time on specific topics. We will cover the topics below, more or less in the order indicated. I will be letting you know in class what to prepare for the next few classes.

- Introduction to computers and computing
- The Python environment
- Values and data types
- Input and output
- Hello World!
- Variables, and assignment
- Control Structures (if, if/else, while, for)
- Functions and modules
- Sequences, lists, strings, sets, tuples, dictionaries and comprehensions
- Working with text
- Files
- Generators
- decorators
- Classes – Object Oriented Programming (if time permits)
- additional material in-between and after the above, as time permits

Requirements

Exams

A midterm and final exam

Texts

Listed on the class website.

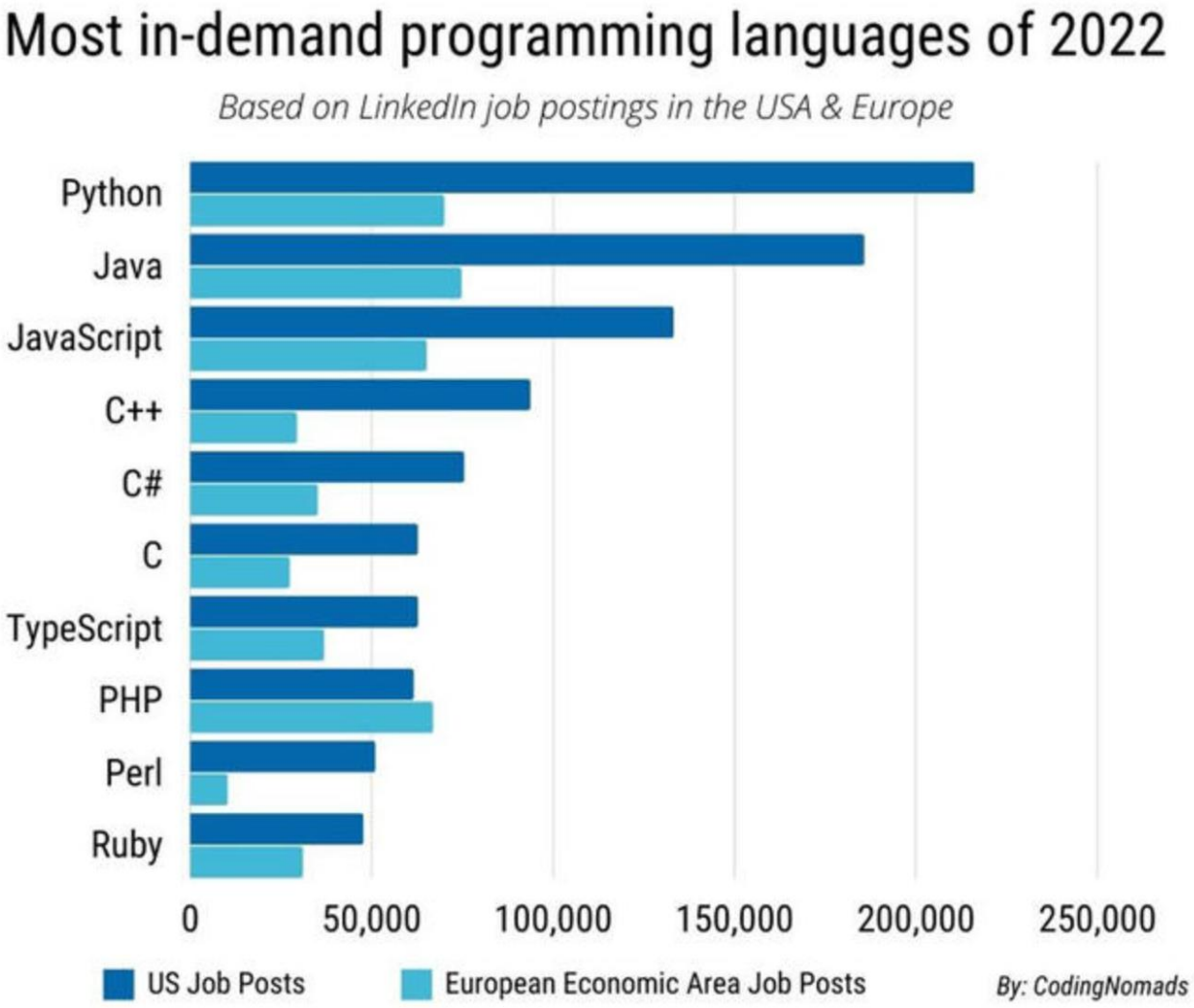
https://venus.cs.qc.cuny.edu/~waxman/cs90/

How to use these notes?

These notes are **not** a substitute for a text book. The notes will provide an outline of important ideas, and a place to work out the problems that we tackle in class. They contain most (but not necessarily all) of the material that we cover in the lecture.

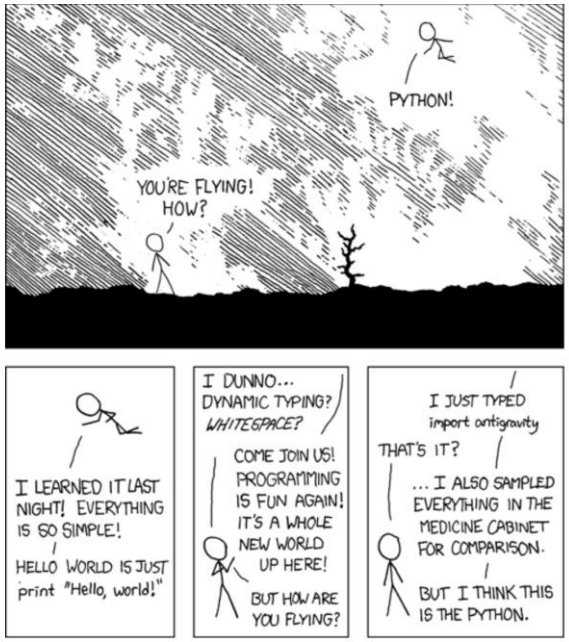
There will be lots of space for you to write answers to question that come up in class. **Use these notes as your notebook to take class notes.**

Important: You are responsible for material that we cover in class but are not in the notes. If you miss any classes, please make sure that you are current.



That was 2022. What about 2024?

https://www.orientsoftware.com/blog/most-popular-programming-languages/



Python

There are many Python Implementations:

1. **CPython**:
    - **Description**: The default and most widely used implementation of Python, written in C.
    - **Key Features**:
        - Most compatible with Python libraries and extensions.
        - Standard interpreter distributed by the Python Software Foundation.
    - **When to Use**:
        - General-purpose use.
        - When you need the broadest compatibility with Python libraries and tools.
2. **PyPy**:
    - **Description**: An alternative implementation of Python, written in Python (RPython).
    - **Key Features**:
        - Just-In-Time (JIT) compiler, which can significantly speed up execution of Python code.
        - Generally faster than CPython for long-running applications.
    - **When to Use**:
        - When you need better performance for your Python code, especially for long-running programs.
        - If you need to run code that is compatible with the standard Python (CPython) but want better performance.

3. **Cython**:
    - **Description**: A programming language that makes writing C extensions for Python as easy as Python itself.
    - **Key Features**:
        - Allows you to write C-like performance code while retaining the syntax and ease of Python.
        - Useful for performance-critical sections of code.
    - **When to Use**:
        - When you need to optimize performance-critical parts of your application.
        - For integrating C libraries with Python.

4. **Jython**:
    - **Description**: An implementation of Python that runs on the Java platform.
    - **Key Features**:
        - Can seamlessly import and use any Java class.
        - Allows you to write Python code that interacts with Java.
    - **When to Use**:
        - When you need to integrate with Java applications or libraries.
        - For leveraging Java's extensive ecosystem while using Python.
5. **IronPython**:
    - **Description**: An implementation of Python running on the .NET framework.
    - **Key Features**:
        - Can use .NET libraries and tools.
        - Ideal for applications that require .NET integration.
    - **When to Use**:
        - When working in a .NET environment and you need to integrate Python with .NET libraries and tools.
6. **MicroPython**:
    - **Description**: A lean and efficient implementation of Python 3 designed to run on microcontrollers.
    - **Key Features**:
        - Designed for microcontrollers with limited resources.
        - Optimized for running on small embedded systems.
    - **When to Use**:
        - For developing applications on microcontrollers and embedded systems.
        - When working with IoT devices that require Python.

Summary:

- **CPython**: General use, compatibility with all Python libraries.
- **PyPy**: Performance boost for long-running applications.
- **Cython**: Performance-critical code sections, integration with C.
- **Jython**: Java integration.
- **IronPython**: .NET integration.
- **MicroPython**: Embedded systems and microcontrollers.

Choosing the right implementation depends on your specific needs, such as performance requirements, compatibility with other languages, or constraints of the target platform.

We will be using CPython, the default and most widely used implementation of Python,

Download Python and install it on your laptop.

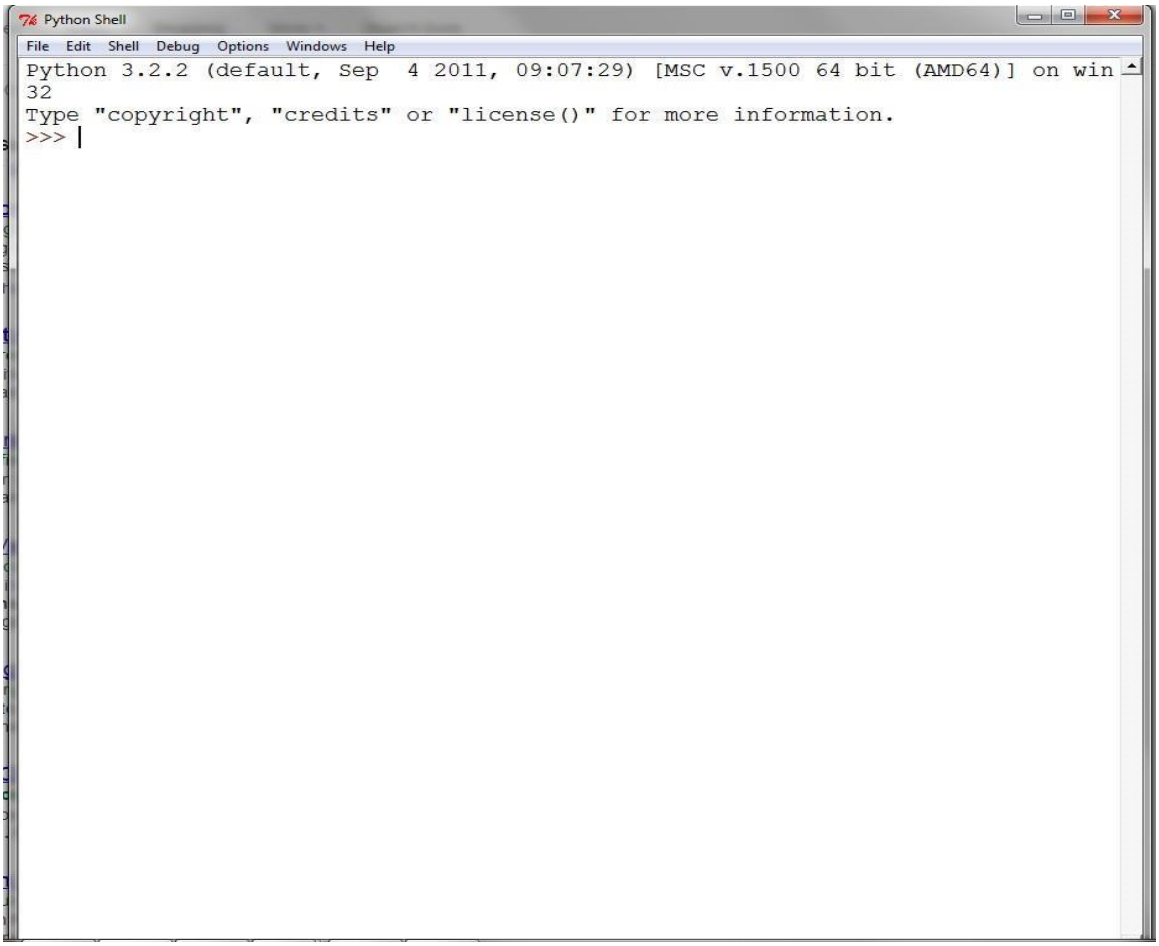
Python can be downloaded for free from the official Python web site:

http://python.org/

We will be using Python 3.9 (or whatever the current latest version is) 3.11. Make sure you download andinstall the correct version (Windows/Mac) and the 32 or 64 bit version.

IDLE – Integrated Development Environment

When Python installs you will be able to access its functionality through the IDLE interface.



We will be accessing Python in three different ways.

1. The interactive shell. This is available when we bring up IDLE. We will make limited use of this, mostly to experiment with various Python constructs and to test out ideas.

2. Python programs. This will be our main focus. A program (sometimes called a "script") is a file containing a sequence of statements in the Python language. We will create these files using IDLE and then "run" the file which will execute the statements in the program.

3. A Python Emulator. This website will allow us to "step through a python program and "see" what is happening in the memory after each instruction is executed. We will use this in class and it is an excellent tool when you are debugging (eliminating errors) your programs. It is available here: http://pythontutor.com/

And now …….

We first examine some Python constructs by using the interactive shell.

Data Types – A Classification of Python's Nouns

More precisely, a data type is thought of as the combination of the nouns and the verbs that they respond to.

Type Category	Type Name	Description
None	`type(None)`	The null object `None`
Numbers	`int`	Integer
	`long`	Arbitrary-precision integer (Python 2 only)
	`float`	Floating point
	`complex`	Complex number
	`bool`	Boolean (`True` or `False`)
Sequences	`str`	Character string
	`unicode`	Unicode character string (Python 2 only)
	`list`	List
	`tuple`	Tuple
	`xrange`	A range of integers created by `xrange()` (In Python 3, it is called `range`.)
Mapping	`dict`	Dictionary
Sets	`set`	Mutable set
	`frozenset`	Immutable set

We will start with some of the basic data types:

- Integers
- Floats
- Strings
- Booleans

.

There will be more later on.

The data types denote sets of the different types of Pythons nouns. **The individual elements of the set, the constants, are the nouns**. For example, One of Python's data types is "integer", i.e. all of the whole numbers. Each particular number is like a noun. It's like in a natural language there is the class of "nouns" (the data type) and individual nouns in that class.

Since one of the main things that we might want to do with Python is mathematical computations, Python provides two kinds of arithmetic "nouns", i.e. numbers, the integers and floats, and operations on them.

When we perform operations on the data we are generation something called an **"expression."** Simply put an expression is something that Python can **evaluate,** that is we can perform the operations and get a value. We will see examples as we look at the specific data types.

Important: In Python, everything is an (first class) object.

What does that mean in practice?

The first data type:

Integer: a whole number, positive, negative or 0. Integers in Python can be arbitrarily long and are written without commas. Other programming languages generally limit the size of integers.
0

56

-897

345678987654323456787665543093764830036455489302002

Note: we can use '_' as a separator to help visualize a long number. This will not effect its representation.

Ex: 123_456_789

Operations on integers:

Python provides the standard mathematical operations on integers, with a twist or two.

1. Addition ➔ 123 + 56

Notice that adding two integers always produces another integer.

2. Subtraction➔ 123 – 56

Notice that subtracting two integers always produces another integer.

3. Multiplication ➔ 123*56

Notice that multiplying two integers always produces another integer.

4. Division There are **two division operations** available for integers: "/" and "//".

The "**/"** operator.

"/" will perform "regular" division so 123/56 ➔ **2.19**, i.e. produces a "floating point" number, (a number with a decimal point) which we will see next.

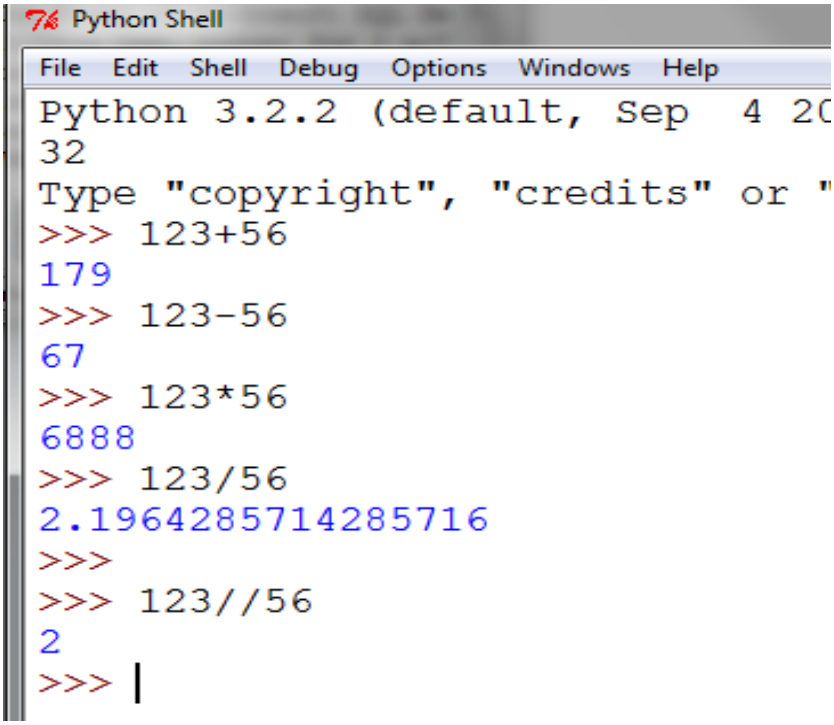
Even when the numbers divide exactly, the **result of "/" will always be a float and contain a decimal point.**

2/1 ➔ **2.0**

The "**//**" operator.

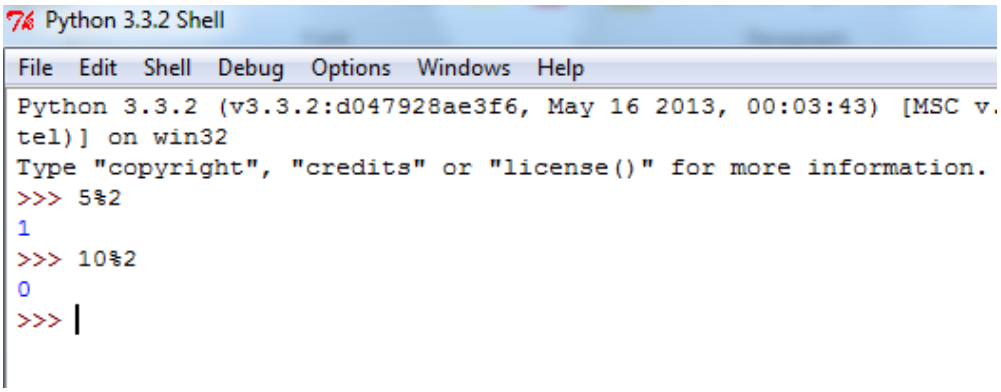
123//56 ➔ 2 Notice that the decimal point and all digits to its right are gone! They have been "**truncated**." It's as if Python took the 2.19 above and "chopped off" the decimal point and everything to its right.

This "//" operation on integers will **always** produce **an integer result**.



5. "%" the remainder function, called **mod**. If a and b are integers then a%b returns (evaluates to) the remainder of a divided by b.

For example:



6. Exponentiation: **

For example: 5**3=125

Operator Precedence (from the word precede)

What is that?

How do we evaluate expressions that contain many operations?

In order of the operator precedence.

Operator	Description
lambda	Lambda Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[index:index]	Slicing
f(arguments ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
`expressions, ...`	String conversion

The order in the table above is from **lowest to highest precedence.**

For example:

Important: Just like in algebra, we can modify the order of evaluation by using parenthesis.

The second data type:

Float: A floating point number is one that contains a decimal point. We saw an example of this above with the "/" operation on integers. There is an important technical difference between integers and floats. As we saw integers can be arbitrarily long (subject to the limitations of your particular computer hardware) and operations that yield integers are always exact. This is not the case with floats. Unlike integers, floats have maximum and minimum sizes.

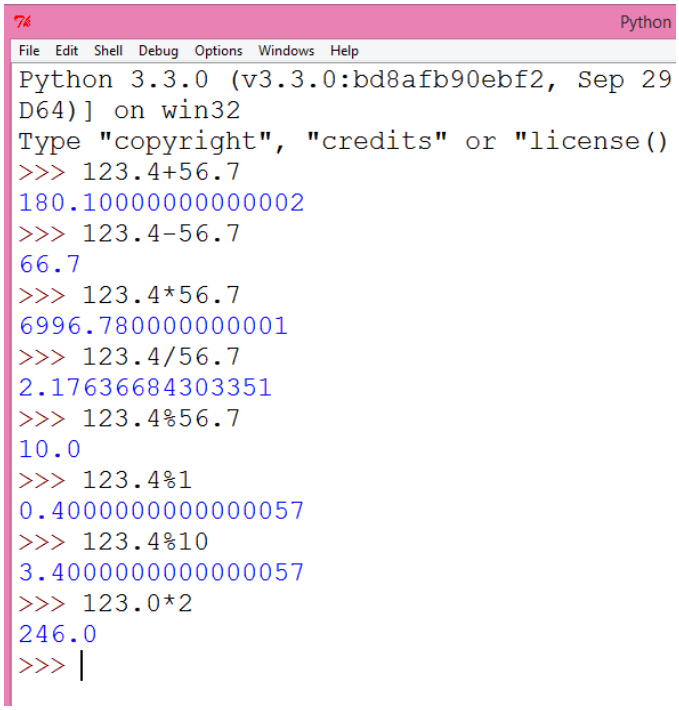
max=1.7976931348623157e+308
min=2.2250738585072014e-308

If any operation yields a float value larger than max, we have an **overflow** condition. Likewise, if it yields a value smaller than min, we have an **underflow** condition, and the values obtained are incorrect.

Additionally, because of how floats are represented in the computer, many values can only be approximated. Many floats are only approximate, 1/3 = 0.3333333333333333.

Operations on floats:

Like it does for the integers, **Python provides the basic arithmetic operations** that you would expect. Python even defines the mod function for floats as the following examples illustrate.

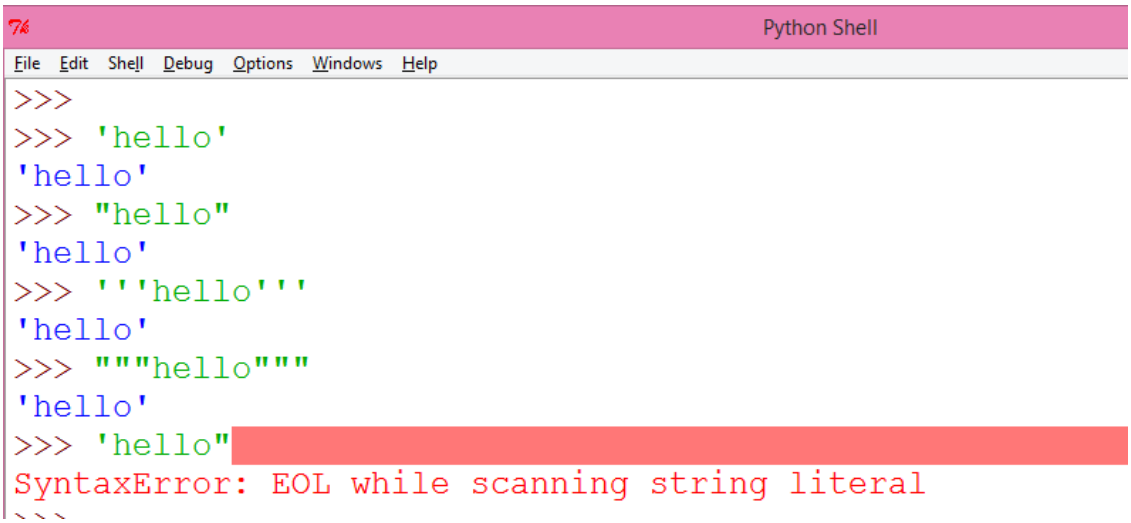


Notice that operations involving two floats produce a float result. This is true even if one of the operand is an integer. So 123.0 * 2 produces a float even though 2 is an integer.

The next two data types are not arithmetic.

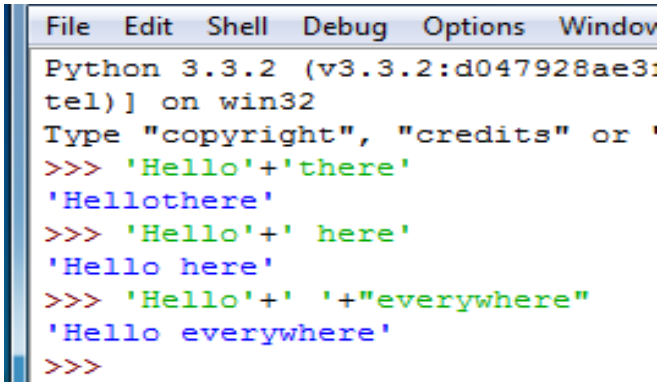
The third data type:

String: A string is a sequence of one or more **characters**. An individual character is also a string in Python. We denote a string by surrounding the character sequence by matching pairs of quotes. The quotes are: (1) ‘, (2)”, (3) three of the first two – ‘’’ or “””. Here are some examples:

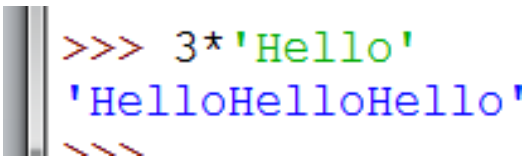


Operations on strings:

Python provides a very rich set of string operations and functions. We will see them later on. For now,we look at one of the most important one**: “+” called concatenation.**



What about 3*’Hello’? Can we “multiply strings?



How come?????? … because just as 3*5 => 5+5+5 (repeated addition) likewise

3* ‘Hello’ => ‘Hello’+ ‘Hello’+ ‘Hello’

Question: What about 0* ‘Hello’ ?

The forth data type:

Boolean: This data type has two values: True and False (spelled just as you see them – first letter caps and others lower case).

In addition … **certain operations yield Boolean values**, specifically, **comparison operations on arithmetic and string expressions yield comparison expressions.**

Arithmetic expressions are expressions that yield numbers. String expressions yield strings ('abc'+'def' ➔ 'abcdef'). We have seen these above.

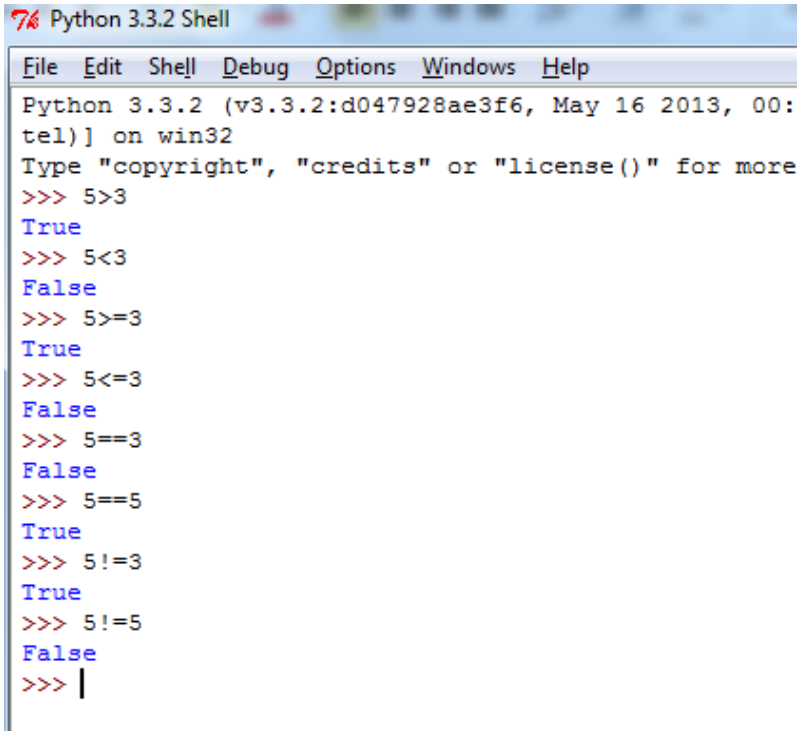
What are the **comparison operations provided by Python**?

Comparison operations

Python provides the following comparison operations:

>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

When we use these to compare the values of arithmetic expressions**, we get a Boolean value**.



We can also use these comparison operations on strings. The value of the comparison depends of the lexicographic ordering (dictionary ordering – which string appears earlier or later in a dictionary) of the strings.

```
>>> 'abc'<'de'
True
>>> 'abc'<'ab'
False
>>> 'd'>'abc'
True
>>>
```

We can **chain the comparison operations in a standard "mathematical" way** and write things like:

x<=y<z where x,y,z are appropriate values.

Comparison expressions may be combined using **Boolean operations:**

and
or
not

The operator `not` yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression `x or y` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> True and 6
6
>>> True or 6
True
>>> False and 6
False
>>> False or 6
6
>>>
```

Look at this:

```
>>> (7>5)*6
6
>>> (7<5)*6
0
```

What is going on? How can we multiply (7>5) which evaluates to True by a number and get a number? Same question with (7<5).

Answer:

There are four more comparison operations provided by Python:

is
is not
in
not in

We will be using these later when we work with sequences and collections of various sorts.

Conversion between data types

Python provides a number of functions that let you convert between different data types:

float()

int()

str()

These are actually conversion constructors.

The function **type(x)** returns the "type', i.e. the kind of object x is.

```
>>> 5
5
>>> type(5)
<class 'int'>
>>> float(5)
5.0
>>> str(5)
'5'
>>> 5.0
5.0
>>> type(5.0)
<class 'float'>
>>> int(5.0)
5
>>> str(5.0)
'5.0'
>>> '5'
'5'
>>> type('5')
<class 'str'>
>>> int('5')
5
>>> float('5')
5.0
>>>
```

We can compare types:

```
>>> type(1)==type(1.0)
False
>>> type(5)==type('5')
False
>>> type(1)==type(5)
True
>>>
```

Output

How do we get Python to output information?

We have just seen the basic "nouns" (=data types) and some operations on them, i.e. item 3 below. We see that Python can get the computer to perform arithmetic and symbolic (e.g. string) manipulation.

Recall:

1. Input
2. Output
3. Memory
4. **Arithmetic – symbolic manipulation and evaluation**
5. Control

What about **output**?

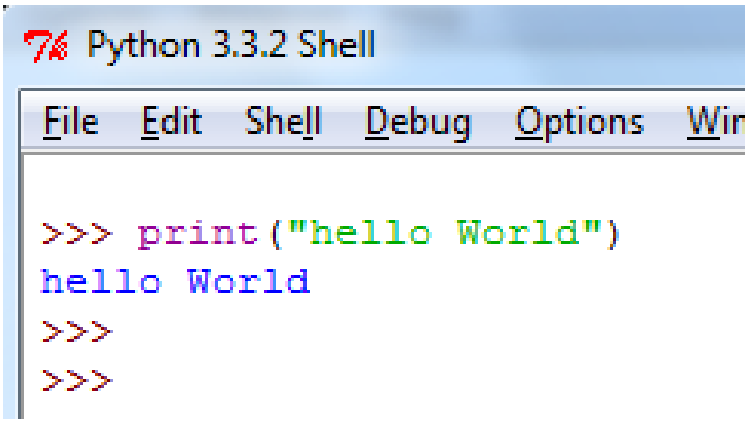
1. Input
2. **Output**
3. Memory
4. Arithmetic – symbolic manipulation and evaluation
5. Control

Question: Where can we output to?

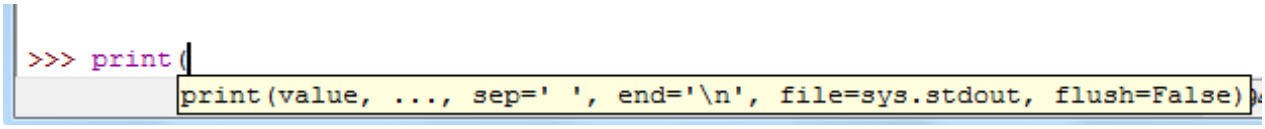
For now, all of our output will be sent to the **monitor** (=screen).

How? We use the **print statement.**

print('Hello world')



Here is the syntax of the print statement.



As soon as I started to type the print command Python shows me the syntax. Here is how to read it.

First of all, **print is a "function."** A function has

- A name (here – **print** is the function name)
- Followed by parenthesis
- And zero or more "arguments" that are "passed in". If there is more than one argument, they are separated from each other by a comma.

What do the arguments mean/control?

value

sep

end

file

flush

What is a default value:

Now:

Print the values 1, 2, 3 all on one line

Print the values 1,2 3 one per line. Do this in two ways.

More examples using print: