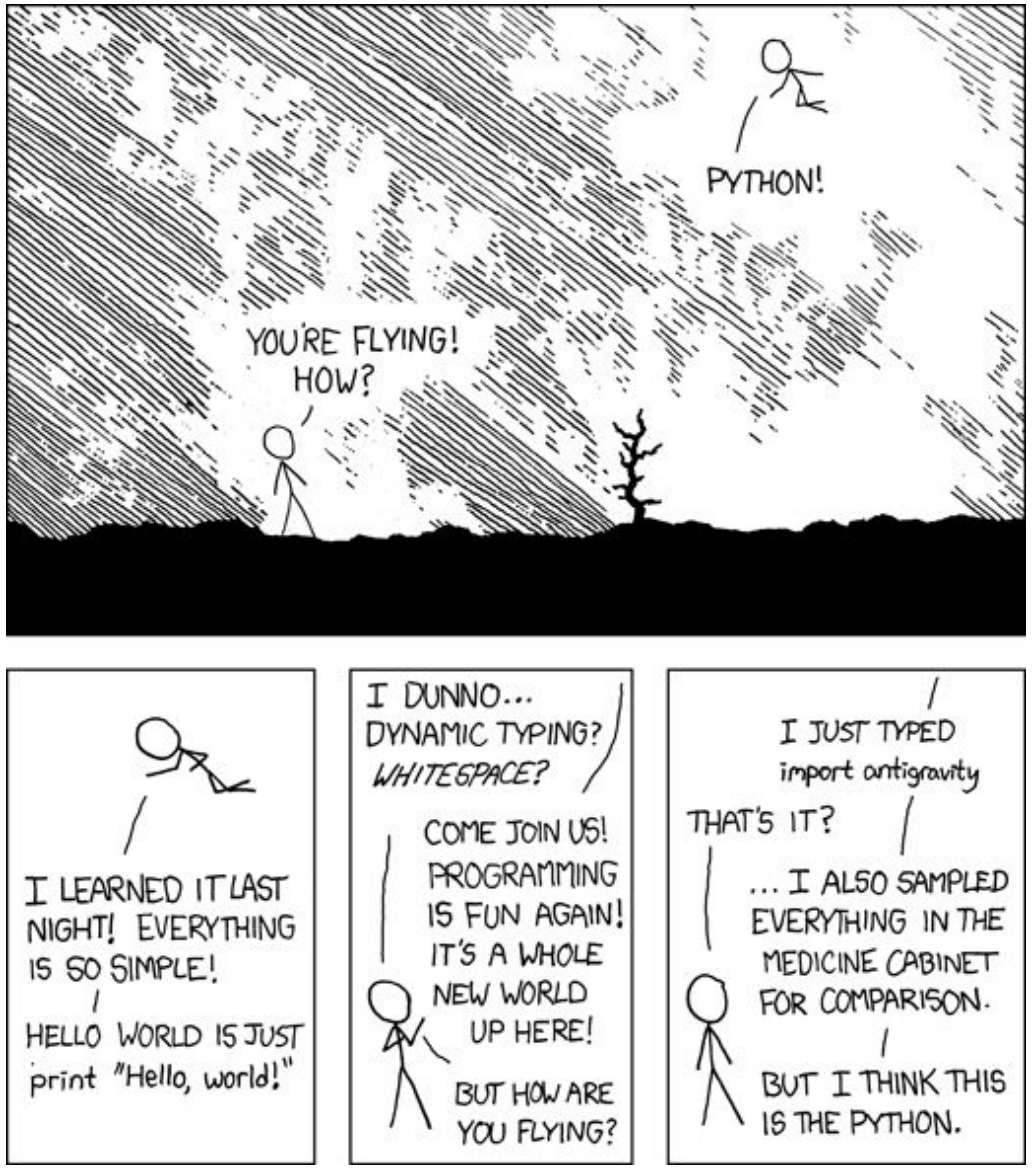
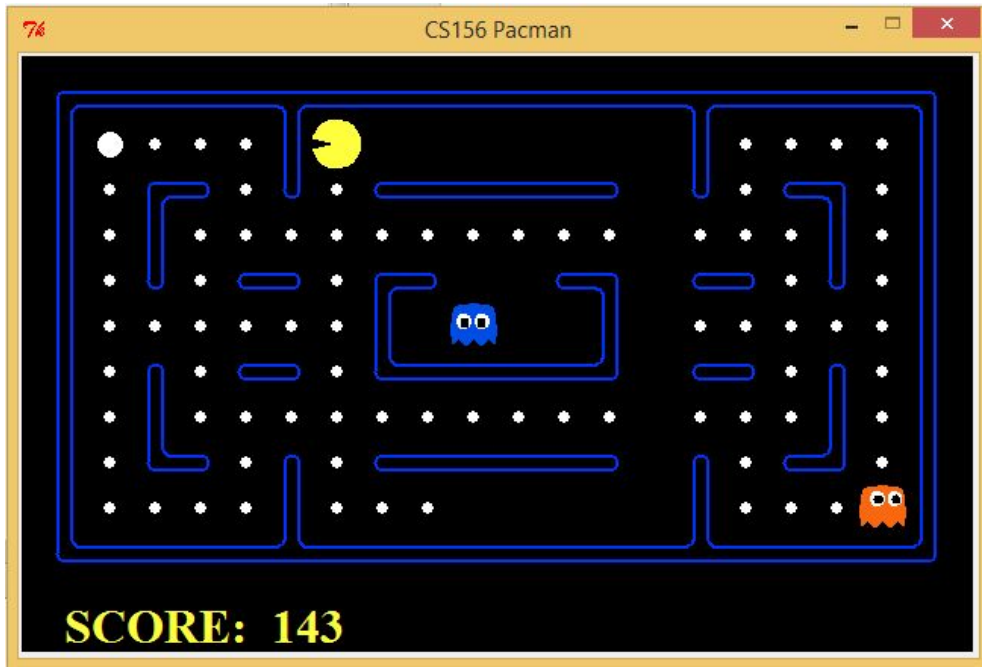
http://xkcd.com/353/



Why Python?

Our programming projects will be based on the Pac-Man AI projects developed at UC Berkeley by John DeNero and Dan Klein.

These projects use Python 2.7.



Pac-Man is a registered trademark of Namco-Bandai Games, used here for educational purposes.

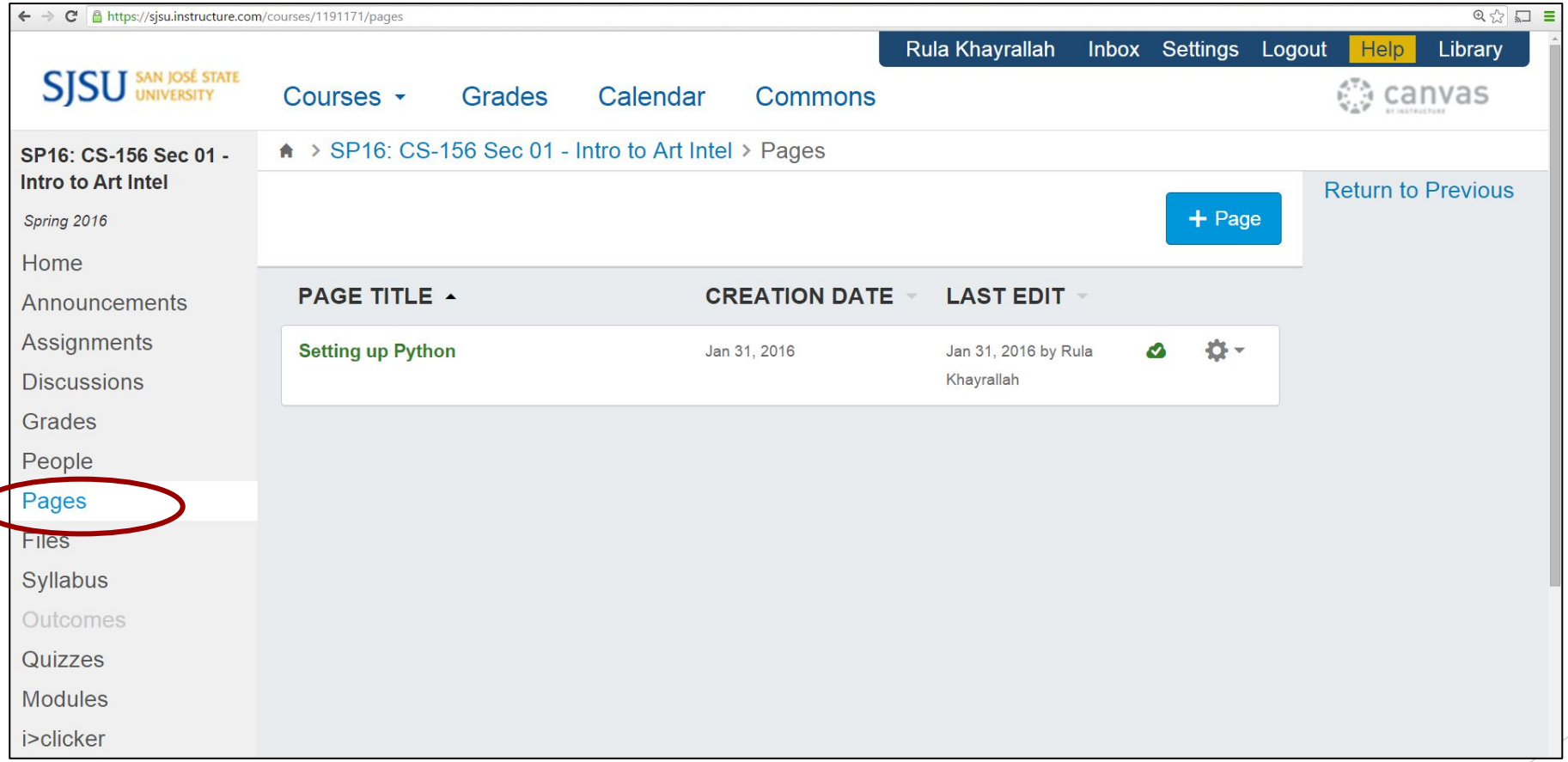
Python

- Python is a fun programming language created in the 90s and named after the comedy series Monty Python.
- It is easier to learn, understand, and remember than most other languages.
- It is **simple, concise yet powerful**.
- Python code is easy to understand, **even for someone who did not write it** and **even if the comments are missing or incorrect.**
- Python programs are typically 3-5 times shorter than equivalent Java programs and they are often 5-10 times shorter than equivalent C++ code.

Setting up Python

- You need to install **Python 2.7** on your personal computer.
- I strongly recommend installing **PyCharm Community Edition**, a free lightweight Python Integrated Development Environment (IDE).
- You'll find all the relevant links and instructions on Canvas.

Setting up Python

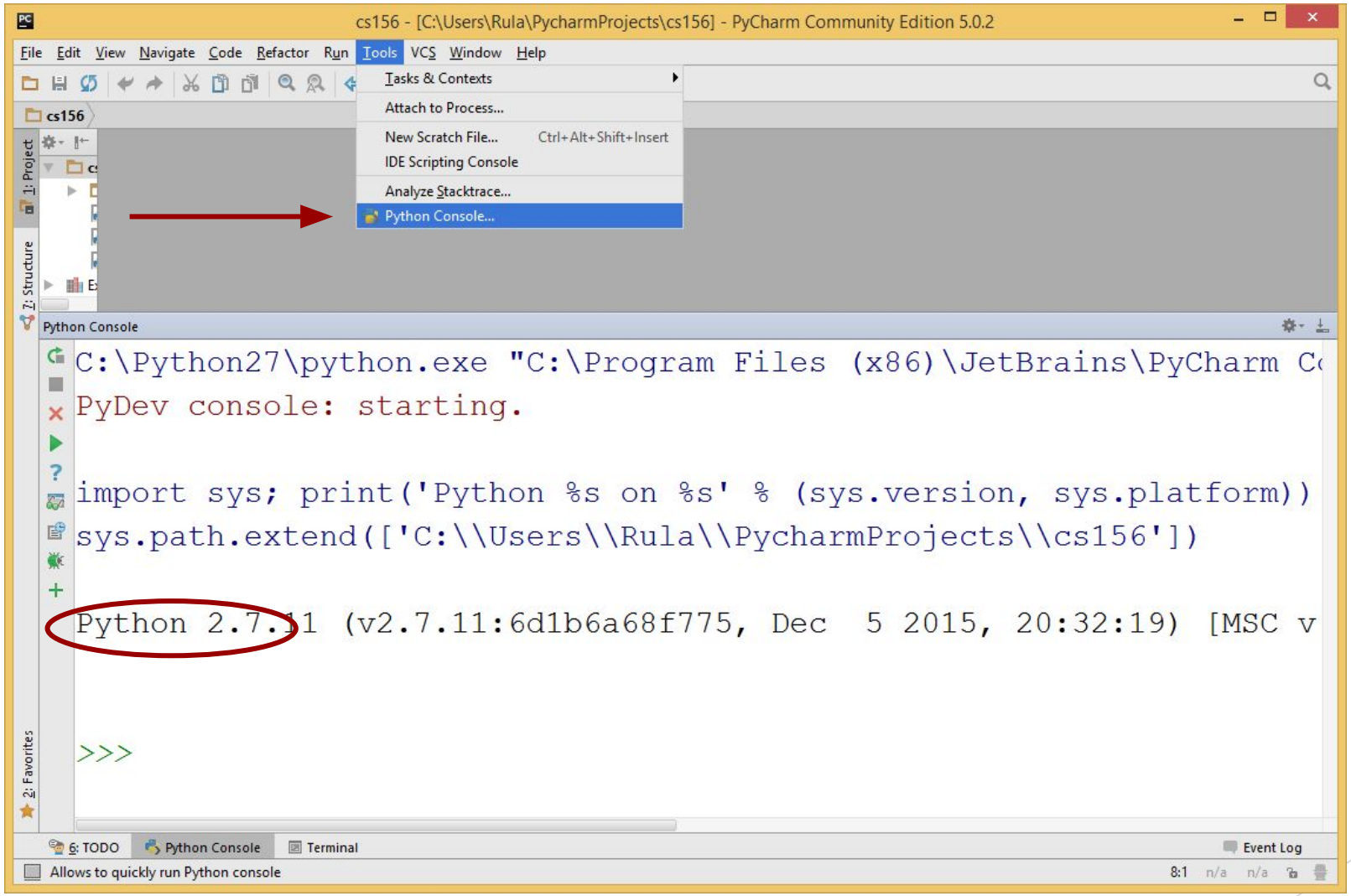


The Python Interpreter

We can use Python interactively by invoking the interpreter. We can do that in several ways:

1. **By starting PyCharm and running the Python Console.**

The Python Interpreter - PyCharm

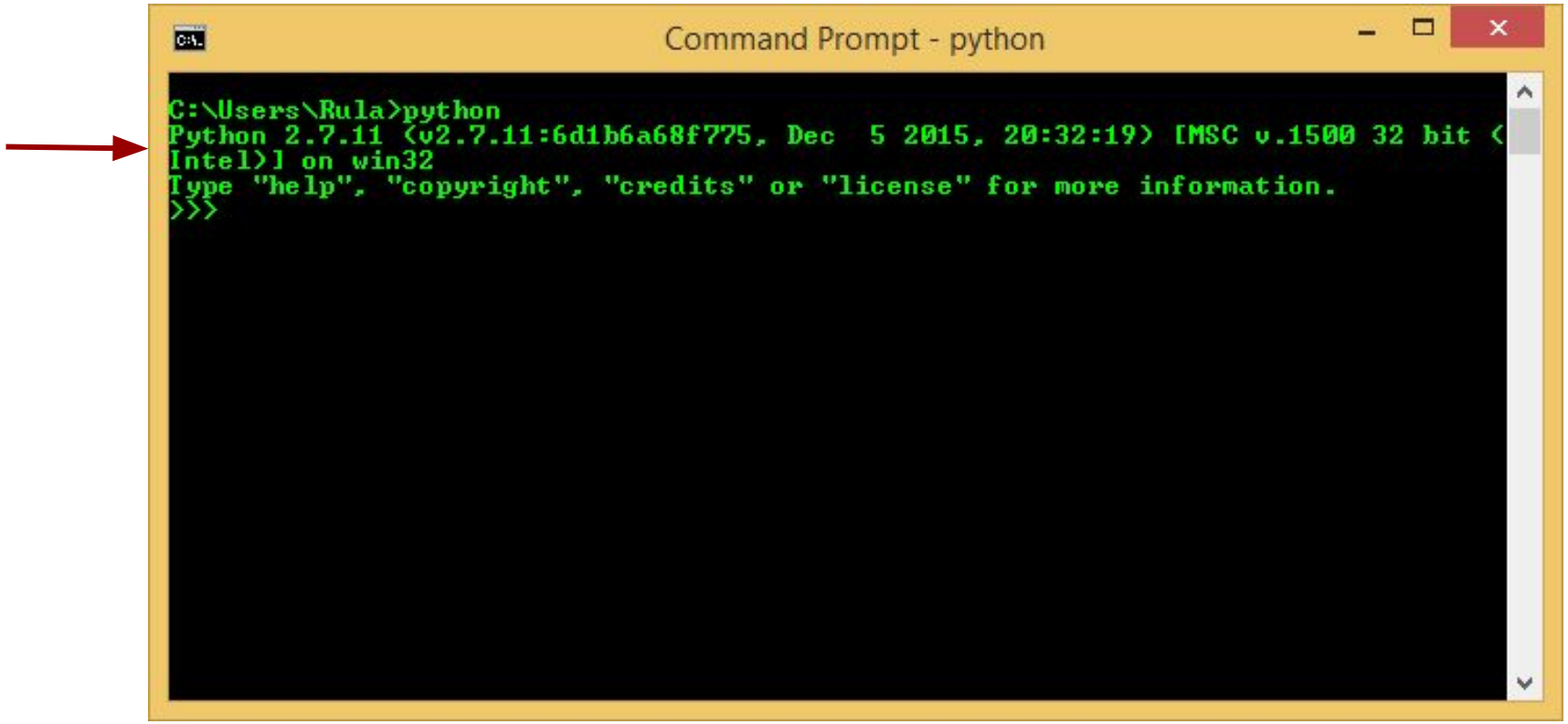


The Python Interpreter

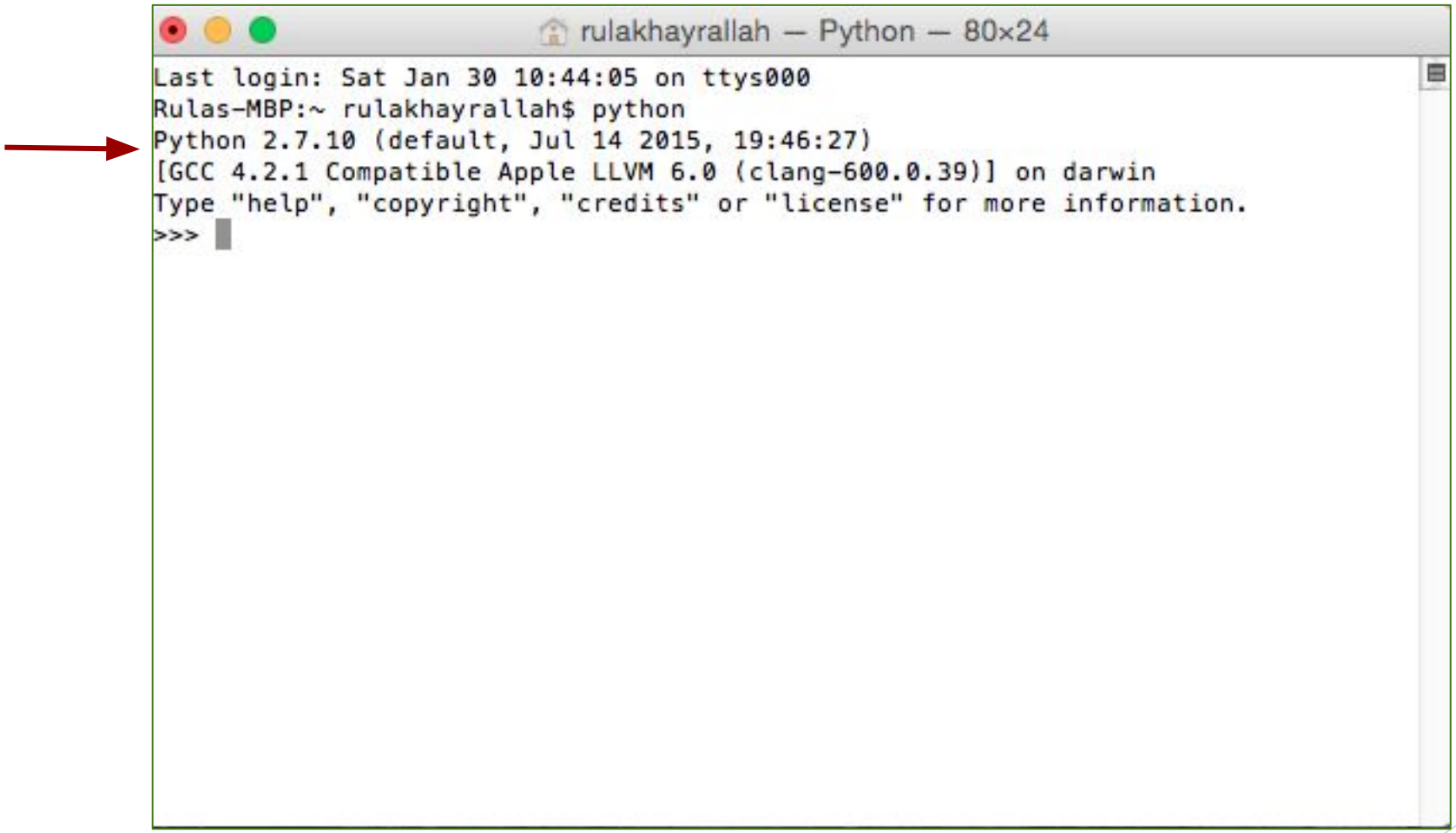
We can use Python interactively by invoking the interpreter. We can do that in several ways:

1. By starting PyCharm and running the Python Console.
2. **By invoking the interpreter from the command prompt or terminal window.**

The Python Interpreter Command Prompt



The Python Interpreter - Terminal

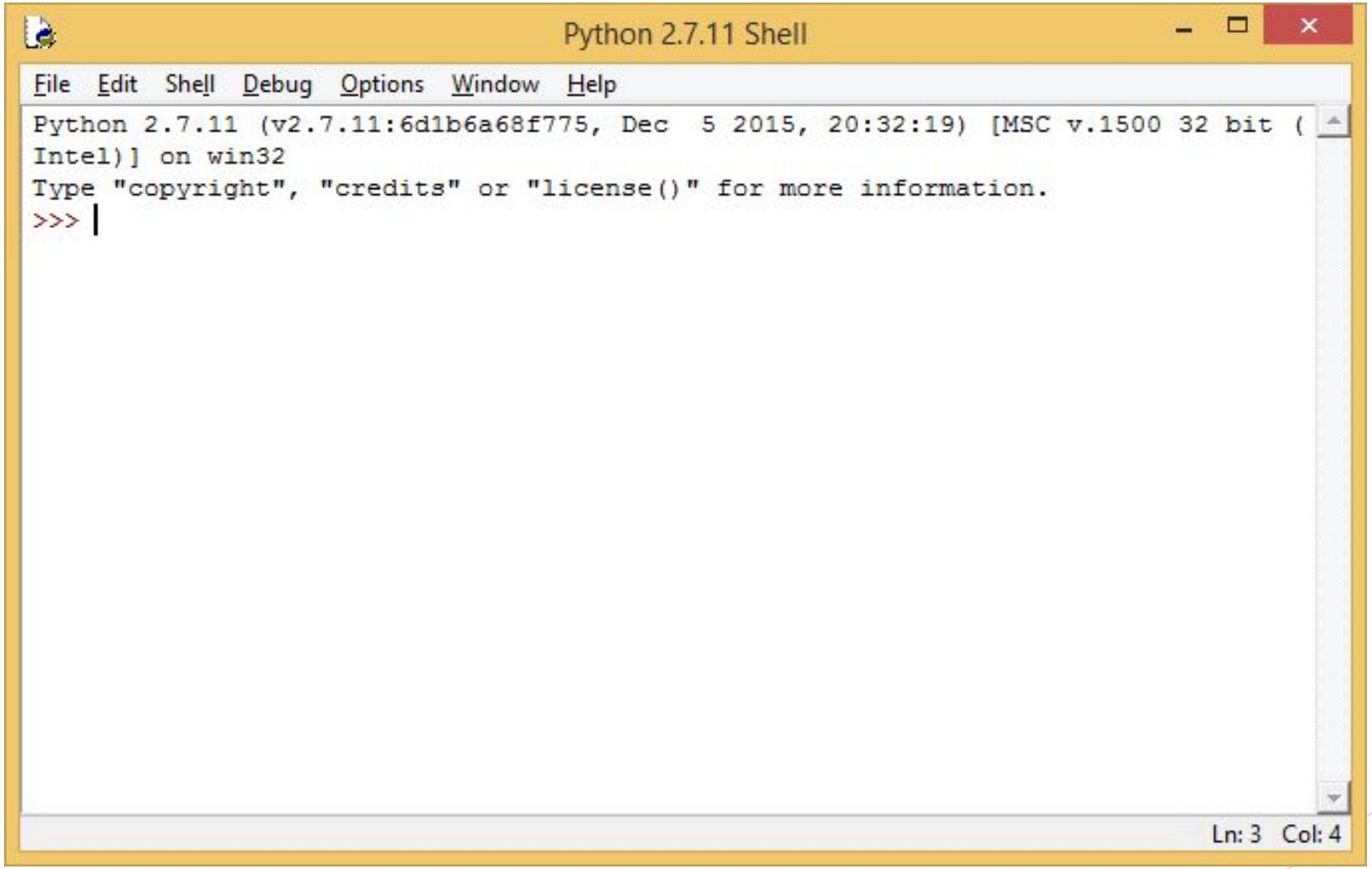


The Python Interpreter

We can use Python interactively by invoking the interpreter. We can do that in several ways:

1. By starting PyCharm and running the Python Console.
2. By invoking the interpreter from the command line/terminal window.
3. **From IDLE, the standard Python development environment.**

The Python Interpreter - IDLE



The Python Interpreter

We can type Python code directly at the interpreter prompt '>>>'. Make sure that you are accessing python 2.7. The version number appears right above the prompt.

Whenever we enter a complete code fragment, it will be executed.

```
>>> 5 + 23
28
>>> print 'Hello World!'
Hello World!
```

Numbers & Operators

Python supports:

- **Integers** such as 9, -2 and 0
- **Floats** such as 5.0 and -4.5
- **Complex** numbers such as complex(1, 2) which denotes 1 + 2j

We can use the conversion functions *int* and *float* to convert from one type to another.

```
>>> float(1)
1.0
>>> int(6.7)
6
```

Numbers & Operators

The basic mathematical operators are built into the Python language:

+ addition

- subtraction

* multiplication

/ division

// floor division

** exponentiation

% remainder

Division in Python 2

In Python 2, the result of dividing two integers is an integer (this is not the case in Python 3).

```
>>> 10 / 2
5
>>> 9 / 2
4
>>> 9.0 / 2
4.5
```

Booleans

Booleans are either **True** or **False** (capital T and F.)

There are 3 logical operators defined on Booleans: **and**, **or**, **not**

Familiar rules of Boolean algebra apply:

not True	False
not False	True
True and True	True
True and False	False
False and True	False
False and False	False
True or True	True
True or False	True
False or True	True
False or False	False

Comparison Operators

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

Comparison Operators

```
>>> 5 > 3
True
>>> 2 <= 1
False
>>> 200.2 < 989.1
True
>>> 0 != -16
True
```

Variables

```
>>> result = True
>>> result
True
>>> type(result)
<type 'bool'>
>>> result = 5
>>> type(result)
<type 'int'>
```

In Python, we don't have to declare a variable type before using it. The variable takes the type of whatever value we assign to it.

Now *result* is assigned the integer value 5. No error is generated. *result* is now of type integer.

Strings

Strings may be enclosed in:

- single quotes: 'Hi'
- double quotes: "Hello"
- triple quotes: '''Howdy'''

Triple quotes allow strings to span multiple lines.

Strings

```
>>> friend = 'Bob'
>>> print "Hello", friend
Hello Bob
>>> type(friend)
<type 'str'>
>>> len(friend)
3
```

Strings

```
>>> language = 'Py' + 'thon'

>>> print language

Python
```

+ just concatenates the two strings 'Py' and 'thon' with no space in between.

Strings

Characters in a string can be accessed using the standard [ ] syntax.
Python uses zero-based indexing.

```
>>> friend = 'Bob'
>>> friend[0]
'B'
>>> friend[1]
'o'
>>> friend[2]
'b'
```

Strings are immutable

Even though we can access individual characters in strings, we cannot modify them.

```
>>> friend = 'Bob'
>>> friend[0] = 'A'
Traceback (most recent call last):
TypeError: 'str' object does not support item assignment
```

String Slices

- A **slice** lets us refer to parts of the strings. Later we'll also see it with lists.
- The slice s[start:end] consists of the characters of the string s **beginning at start and extending up to but not including end.**

String Slices

A	r	t	i	f	i	c	i	a	l
0	1	2	3	4	5	6	7	8	9

```
>>> course = 'Artificial'
```

course[0:3] -- chars starting at index 0 and extending up to but not including index 3

```
>>> course[0:3]
'Art'
```

String Slices

A	r	t	i	f	i	c	i	a	l
0	1	2	3	4	5	6	7	8	9

```
>>> course = 'Artificial'
```

course[4:] -- omitting the end index defaults to the end of the string

```
>>> course[4:]
'ficial'
```

String Slices

A	r	t	i	f	i	c	i	a	l
0	1	2	3	4	5	6	7	8	9

```
>>> course = 'Artificial'
```

course[:4] -- omitting the start index defaults to the start of the string

```
>>> course[:4]
'Arti'
```

String Slices

A	r	t	i	f	i	c	i	a	l
0	1	2	3	4	5	6	7	8	9

```
>>> course = 'Artificial'
```

course[:] -- omitting both always gives us a copy of the whole thing

```
>>> course[:]
'Artificial'
```

String Slices

```
 A r t i f i c i a l
-10-9-8-7-6-5-4-3-2-1
```

```
>>> course = 'Artificial'
```

course[-1] -- negative index numbers count back from the end of the string

```
>>> course[-1]
'l'
```

Comments

A comment starts with the hash character (#) and extends to the end of the line.

```
# this whole line is ignored by Python
print "Hello"     # everything from the # on is ignored by Python
```

Docstrings

Documentation strings (or docstrings) are used to create easily-accessible documentation that is picked up and displayed by the *help* function.

We can add a docstring to a module, function or class by adding a string **as the first statement.**

The convention is to use **triple-quoted strings**, because it makes it easier to add documentation spanning multiple lines.

Our First Python Program

- In PyCharm, we create a new project
- We make sure the project is selected and click on File -> New, and select File.
- We'll name our program firsthello.py.
- Python programs have to be created with the file extension py.

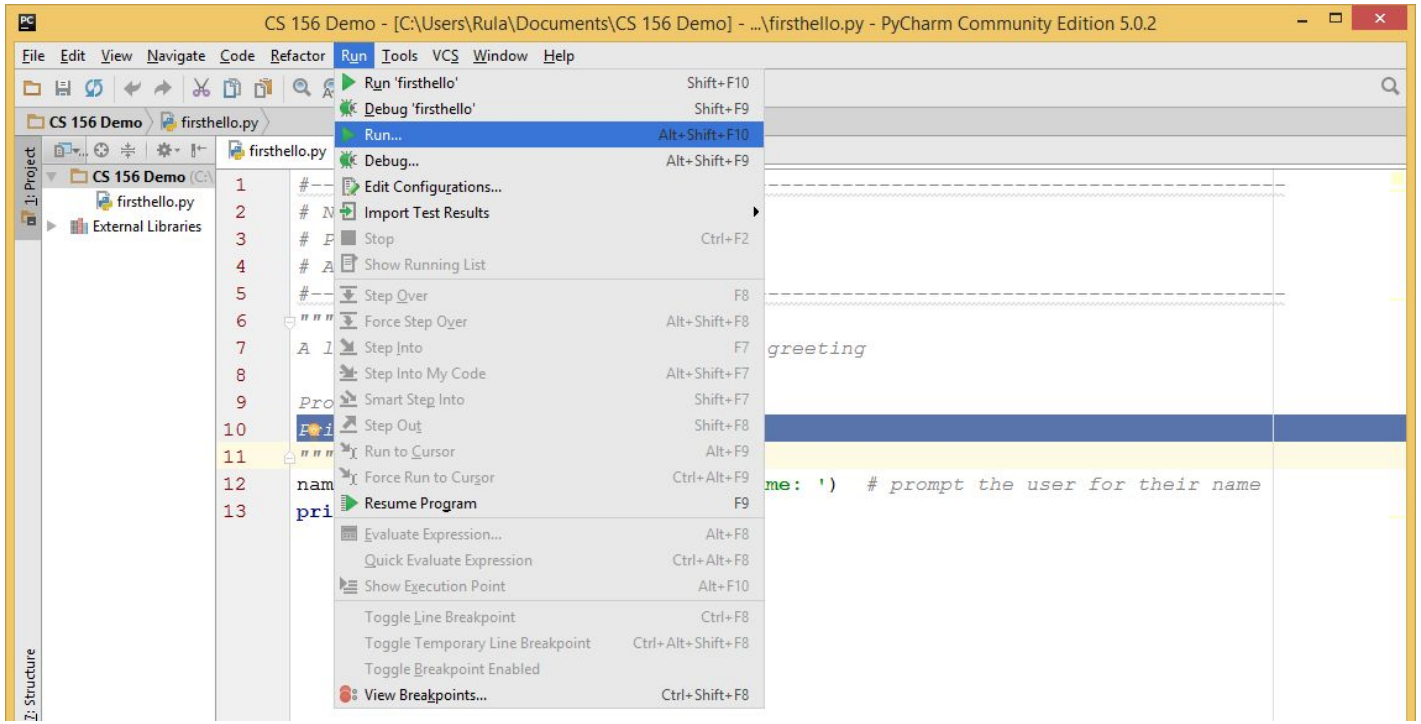
Our First Python Program

```
"""
A little Python program that prints a greeting

Prompt the user for their name.
Print a customized Hello message.
"""
name = raw_input('Please enter your name: ')   # prompt user
print 'Hello', name            # print a personalized greeting
```

Our First Python Program

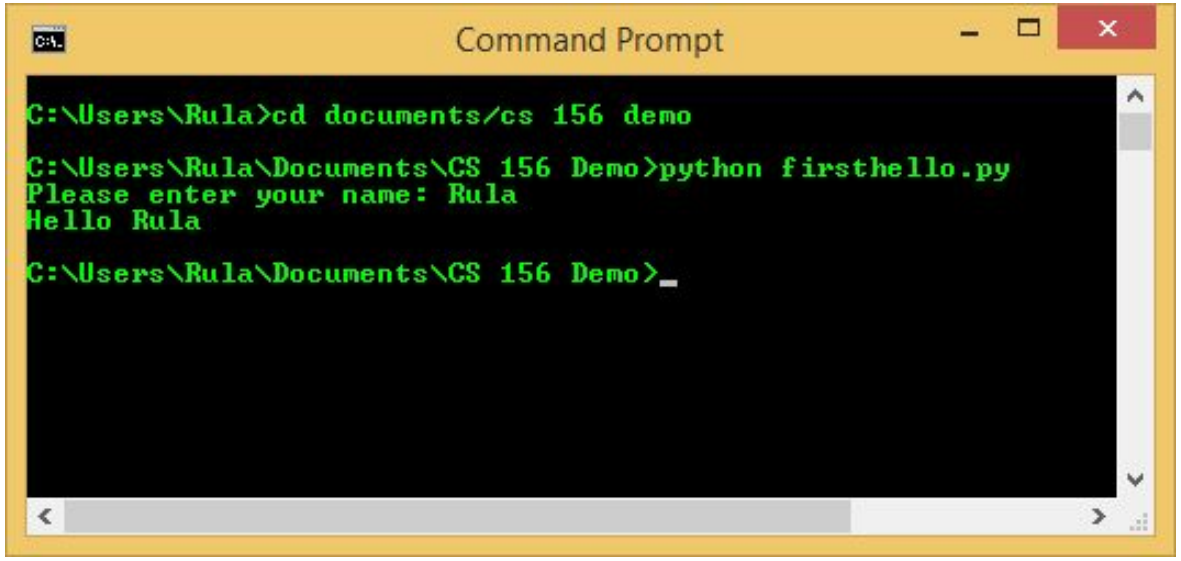
We can run our program from PyCharm by clicking on the Run tab then selecting Run firsthello.py.



Our First Python Program

We can also run our program from a command prompt or terminal window by:

- navigating to the folder containing our program
- typing: python firsthello.py



Built-in Data Structures: Lists

Python lists are similar to arrays in other programming languages.

The elements of the list don't have to have the same type.

```
>>> my_list = ['Shakespeare', 500, 3.2, True]
>>> len(my_list)
4
```

Lists

List indexing is similar to string indexing. We use the square brackets [ ] to access the items in the list, with the **first item at index 0 and the last item at index -1.**

```
>>> my_list = ['Shakespeare' , 500 , 3.2, True]
>>> my_list[0]
'Shakespeare'
>>> my_list[3]
True
```

Lists

Negative index numbers count back from the end of the list:

```
>>> my_list = ['Shakespeare' , 500 , 3.2, True]
>>> my_list[-1]
True
>>> my_list[-2]
3.2
```

Lists

We can get a **slice** of the list.

my_list[start:stop] is a sublist that is made up of the items at *start* up to but not including the item at *stop*.

```
>>> my_list = ['Shakespeare' , 500 , 3.2, True]

>>> my_list[1:3]

[500, 3.2]
```

Lists

```
>>> my_list = ['Shakespeare' , 500 , 3.2, True]
```

Omitting either index defaults to the start or end of the list.

```
>>> my_list[1:]
[500, 3.2, True]
>>> my_list[:-1]
['Shakespeare', 500, 3.2]
```

Omitting both indexes gives us all the items in the list.

```
 >>> my_list[:]
['Shakespeare', 500, 3.2, True]
```

Concatenating Lists

We can use the + operator to concatenate lists as we did for strings.

```
>>> my_list = ['Shakespeare', 500, 3.2, True]
>>> new_list = my_list + ['red','white','blue']
['Shakespeare', 500, 3.2, True, 'red', 'white', 'blue']
>>> my_list
['Shakespeare', 500, 3.2, True]
```

The original list, my_list is unchanged.

Modifying Lists

Unlike strings, which are immutable**, it is possible to change individual elements of a list.**

```
>>> my_list = ['Shakespeare', 500, 3.2, True]
>>> my_list[2] = 0
>>> my_list
['Shakespeare', 500, 0, True]
```

Nested Lists

We can also create lists that contain other lists:

```
>>> exams = [85, 90]
>>> grades = [100, 98, exams, 85]
>>> grades
[100, 98, [85, 90], 85]
```

Membership Test

We can test if an item is in the list using **in** and **not in**:

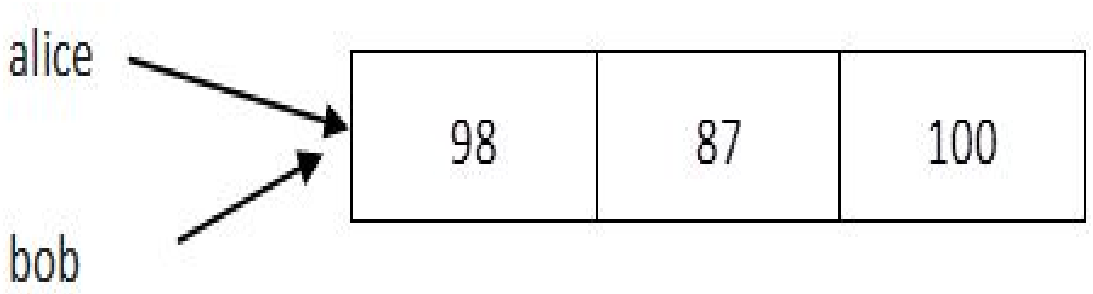
```
>>> grades = [100, 98, [85, 90], 85]
>>>  100 in grades
True
>>> 0 not in grades
True
>>> 90  in grades
False
```

Copying a List

Assignment with an **= sign on lists does not make a copy**. Instead, **the assignment makes the two variables point to the same list in memory.**

Copying a List

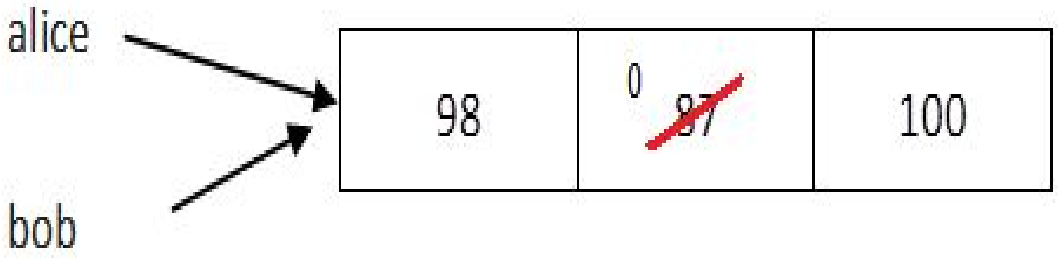
```
>>> alice = [98, 87, 100]
>>> bob = alice
>>> alice
[98, 87, 100]
>>>  bob
[98, 87, 100]
```



Copying a List

What happens if we now change one list item in alice?

```
>>>  alice[1] = 0
>>> alice
[98, 0, 100]
>>>  bob
[98, 0, 100]
```



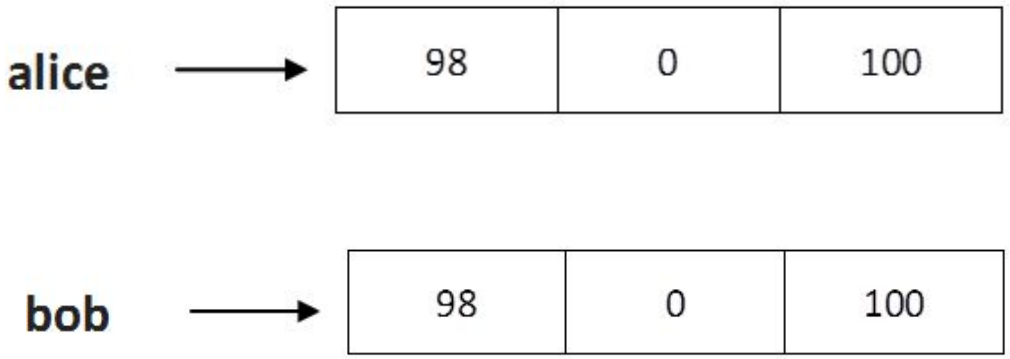
Copying a List

What if we wanted bob to have a **different copy** of the list, one that initially has the same values as alice but that is not affected by future changes to alice?

Copying a List

If the list we are copying **is not nested**, the following **slice assignment** will work.

```
>>> alice = [98, 0, 100]
>>>bob = alice[:]
```



Sorting a list

The built-in function *sorted* returns a sorted list:

```
>>> grades = [85, 60, 100, 95, 75]
>>> sorted(grades)
[60, 75, 85, 95, 100]
```

min, max and sum

```
>>> grades = [85, 60, 100, 95, 75]
>>> min(grades)
60
>>> max(grades)
100
>>> sum(grades)
415
```

List Methods - append

>>> grades = [85, 60, 100, 95, 75]

append is a list method that adds **a new element to the end of a list.**

>>> grades**.append(100)**

>>> grades

[85, 60, 100, 95, 75, **100**]

The original list *grades* has been modified.

List Methods - reverse

```
>>> grades
[85, 60, 100, 95, 75, 100]
```

reverse rearranges the elements in reverse order.

```
>>> grades.reverse()
>>> grades
[100, 75, 95, 100, 60, 85]
```

List Methods - insert

>>> grades

[60, 75, 85, 95, 100, 100]

insert adds an element at a given index position. To insert 90 at index position 0, we write:

>>> grades**.insert(0, 90)**

>>> grades

[**90**, 60, 75, 85, 95, 100, 100]

List Methods - pop

If we know the index of the element we want, we can use **pop**: it **modifies** the list and **returns** the element that was removed.

```
>>> grades
[90, 60, 75, 85, 95, 100, 100]
>>> grades.pop(2)
75
>>> grades
[90, 60, 85, 95, 100, 100]
```

75 is no longer an item in grades.

List Methods - pop

If we don't provide an index, pop **deletes and returns the last element**.

```
>>> grades
[90, 60, 85, 95, 100, 100]
>>> grades.pop()
100
>>> grades
[90, 60, 85, 95, 100]
```

List Methods - remove

If we know the value we want to remove but not the index, we can use remove:

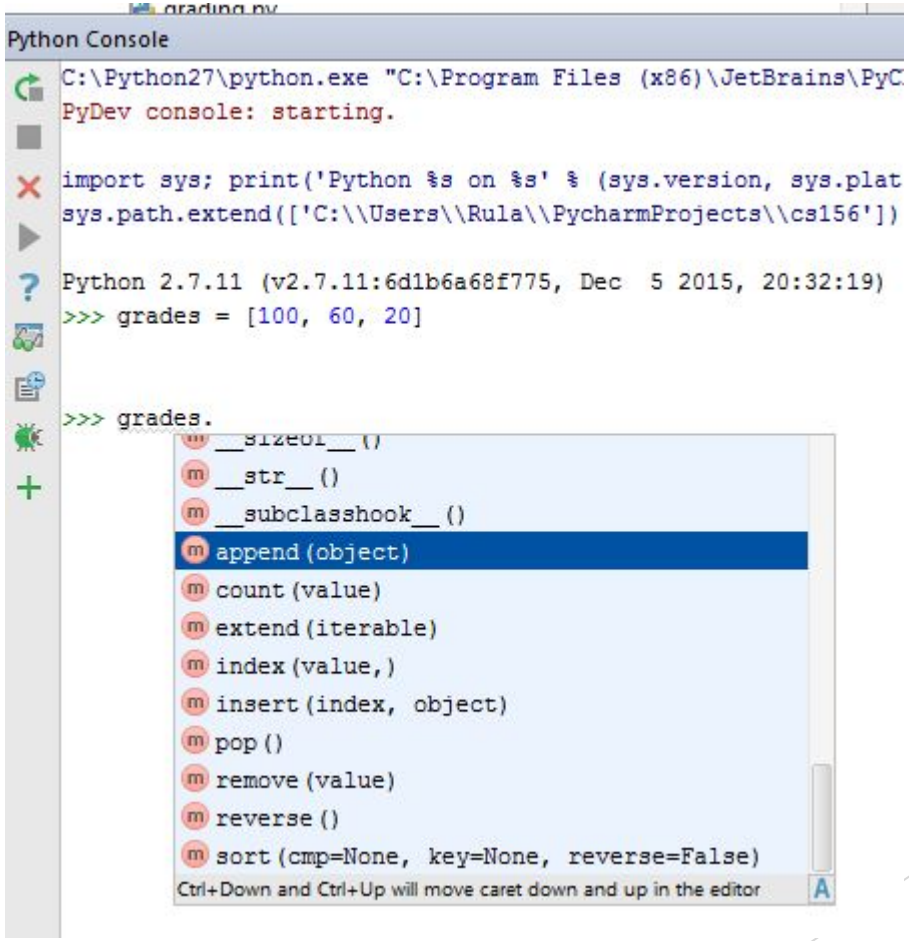
```
>>> grades = [90, 85, 95, 100, 60, 100]
>>> grades.remove(100)
>>> grades
[90, 85, 95, 60, 100]
```

Only the first instance of the given element is removed.

List Methods

One way to see all the methods available on lists is to use code completion in the interpreter shell or the editor.

Typing *grades* followed by a dot brings up a list of all the methods available on grades.



List Methods

We can also use *dir* to see all the methods available on a type:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getslice__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__setslice__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

List Methods

We can also use *help* to get information about a given method:

```
>>> help(list.pop)
Help on method_descriptor:


pop(...)
    L.pop([index]) -> item -- remove and return item at index
(default last).
    Raises IndexError if list is empty or index is out of range.
```

Range

range is a built-in function that returns a list of integers.

range(stop) -> returns a list of integers from 0 up to but not including *stop*.

```
>>> range(3)
```

```
[0, 1, 2]
```

range(start, stop) -> returns a list of integers from *start* up to but not including *stop*.

```
>>> range(1, 5)
```

```
[1, 2, 3, 4]
```

Range

range(start, stop, step) -> returns a list of integers from *start* up to but not including *stop* incrementing by *step*.

Note that *start*, *stop* and *step* have to be **integers**.

```
>>> range(5, 52, 5)
```

[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

Conditional Statements

```
name = raw_input('Please enter your name: ')
if name:
    print 'Hello', name
else:
    print 'Hello Friend'
```

```
if condition:
    Indented Block
else:
    Indented Block
```

The condition is always followed by a colon and the else keyword (if present) is always followed by a colon.

Conditional Statements

```
name = raw_input('Please enter your name: ')
if name:     # equivalent to: if name != '':
    print 'Hello', name
else:
    print 'Hello Friend'
```

In Python we use the boolean interpretation of strings, lists and other sequences in conditions. The boolean interpretation of an empty sequence is False. The boolean interpretation of a non-empty sequence is True.