

Simple Python

Jerry Cain
CS 106AX

October 21, 2024

slides leveraged from those constructed by Eric Roberts

Python Data Types

- Python includes three data types for numbers: the data type `int` for *integers* or whole numbers, the data type `float` for *floating-point numbers* containing a decimal point, and the data type `complex` for *complex numbers*, which are beyond the scope of this course. Much of what you know from JavaScript about integers and floating-point numbers applies to Python as well.
- Python also provides string support in much the same way that JavaScript does. Strings are immutable objects in both JavaScript and Python, and Python includes many of the same string methods you've seen with JavaScript strings.
 - The biggest differences between JavaScript and Python strings are notational, particularly with substrings. The differences are so pronounced that we'll spend all of Wednesday addressing them.

Arithmetic Expressions

- Like most languages, Python specifies computation in the form of an *arithmetic expression*, which consists of *terms* joined together by *operators*.
- Each term in an arithmetic expression is one of the following:
 - An explicit numeric value, such as 2 or 3.14159265
 - A variable name that serves as a placeholder for a value
 - A function call that computes a value
 - An expression enclosed in parentheses
- The operators include the conventional ones from arithmetic, along with a few that are somewhat less familiar:

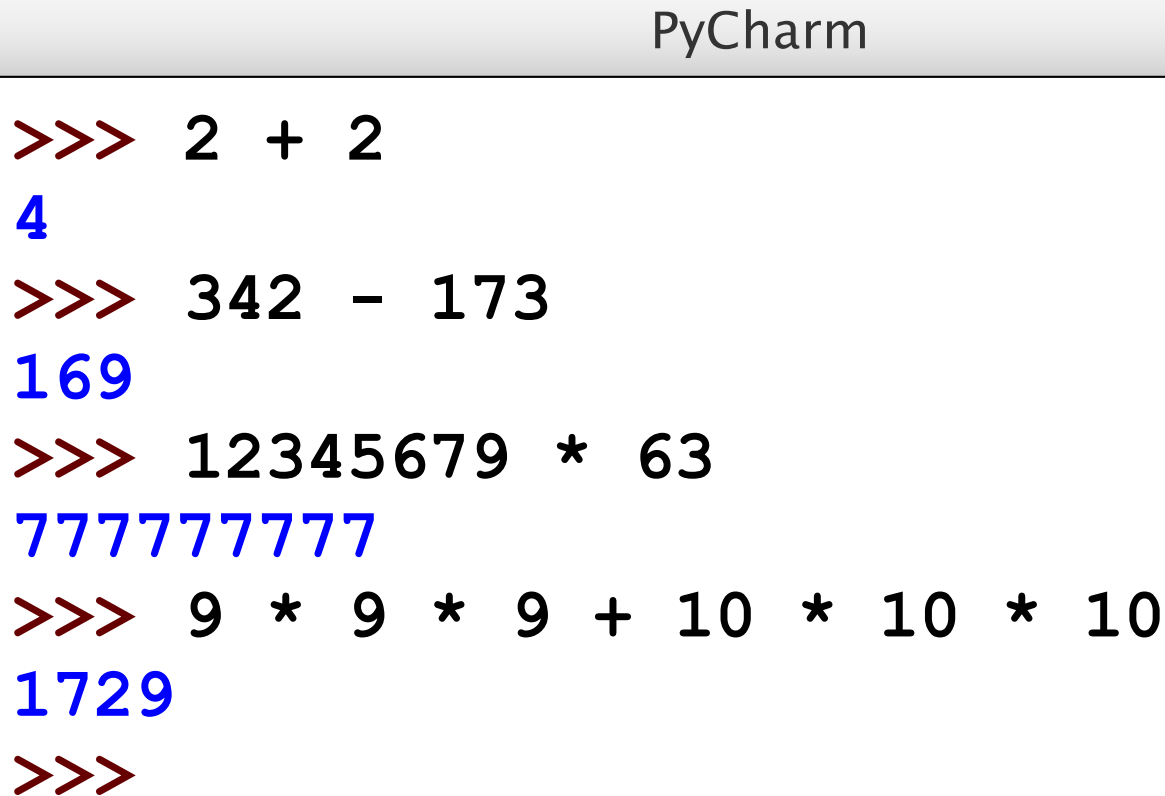
+	Addition	//	Quotient (floor division)
-	Subtraction	%	Remainder
*	Multiplication	**	Exponentiation
/	Division (exact)		

Python Division and Exponentiation

- Python has three operators that involve division:
 - The `/` operator computes the exact result of division, so that the expression `6 / 4` has the value 1.5. Applying the `/` operator always produces a floating-point value.
 - The `//` operator implements *floor division*, which is the result of exact division rounded down to the next smaller integer. The expression `6 // 4` produces the value 1, and `-6 // 4` produces the value -2.
 - The `%` operator behaves the same in Python and JavaScript. As before, we assume the first operand is nonnegative and the second operand is positive, so that `6 % 4` is 2 and `0 % 11` is 0.
- The `**` operator doesn't appear in JavaScript, but it does in Python. It implements exponentiation, so that `3 ** 4` evaluates to 81.
- The other operators all behave as you'd expect.

Using the PyCharm Interpreter

- The easiest way to get a sense of how arithmetic expressions work in Python is to enter them into the [PyCharm interpreter](#), which you'll be using for the Python segment of the course.



A screenshot of the PyCharm interpreter window. The title bar at the top is labeled "PyCharm". The window contains a series of Python commands and their outputs. Each command is preceded by a prompt ">>>" and the output is displayed on the line immediately following. The commands and outputs are: "2 + 2" resulting in "4", "342 - 173" resulting in "169", "12345679 * 63" resulting in "777777777", and "9 * 9 * 9 + 10 * 10 * 10" resulting in "1729". The final line shows the prompt ">>>" without an output.

```
PyCharm
>>> 2 + 2
4
>>> 342 - 173
169
>>> 12345679 * 63
777777777
>>> 9 * 9 * 9 + 10 * 10 * 10
1729
>>>
```

Python Variables and Assignment

- In Python, you create a variable simply by assigning it a value in an *assignment statement*, which has the general form:

```
variable = expression
```

- As in most languages, including JavaScript, the effect of an assignment statement is to compute the value of the expression on the right side of the equal sign and assign that value to the variable that appears on the left, as with:

```
total = total + value
```
- Note you need **not** precede the variable name with the **let** keyword. If the Python interpreter encounters an identifier it's not seen prior, it assumes a new variable is coming into scope.
- Python includes shorthand notation like **+=**, but not ++ or -- !

Naming Conventions

- In Python, names used for variables, functions, and classes are called *identifiers*. Identifier names are composed of letters, digits, and underscores and cannot begin with a digit.
- Programmers use a variety of different conventions to mark word boundaries within a complex identifier name:
 - *Snake case* uses underscores, as in **number_of_students**.
 - *Camel case* embeds uppercase, as in **numberOfStudents**.
- CS 106AX and the Python reader use these conventions:
 - Variable and function names use camel case and begin with a lowercase letter.
 - Constant names are written entirely in uppercase and use the underscore character, as in **MAX_HEADROOM**.
 - Class names and function names designed to be run as programs use camel case and begin with an uppercase letter.

Multiple Assignment

- Python, unlike JavaScript, supports *multiple assignment*, where the left side of an assignment consists of a list of variables and the right side consists of a list of expressions, as with:

`x, y, z = 5, 12, 13`

which sets **`x`** to 5, **`y`** to 12, and **`z`** to 13.

- All the values on the right side are computed before any assignments are performed. For example, the statement

`a, b = b, a`

exchanges the values of the variables **`a`** and **`b`**. JavaScript doesn't allow this.

Functions

- A *function* is a sequence of statements that has been bundled together and given a name, which makes it possible to invoke the complete set of statements simply by supplying that name.
- A function typically takes information from its caller in the form of arguments, performs some computation involving the values of the arguments, and then returns a result to the caller, which continues from the point of the call.
- This notion that functions exist to manipulate information and return results makes functions in programming like functions in mathematics, which is the historical reason for the name.
- All of this is a repeat of how functions generally operate in JavaScript (and virtually all other languages).

Functions in Mathematics

- The graph at the right shows the values of the function

$$f(x) = x^2 - 5$$

- Plugging in a value for x allows you to compute the value of $f(x)$, as follows:

$$f(0) = 0^2 - 5 = -5$$

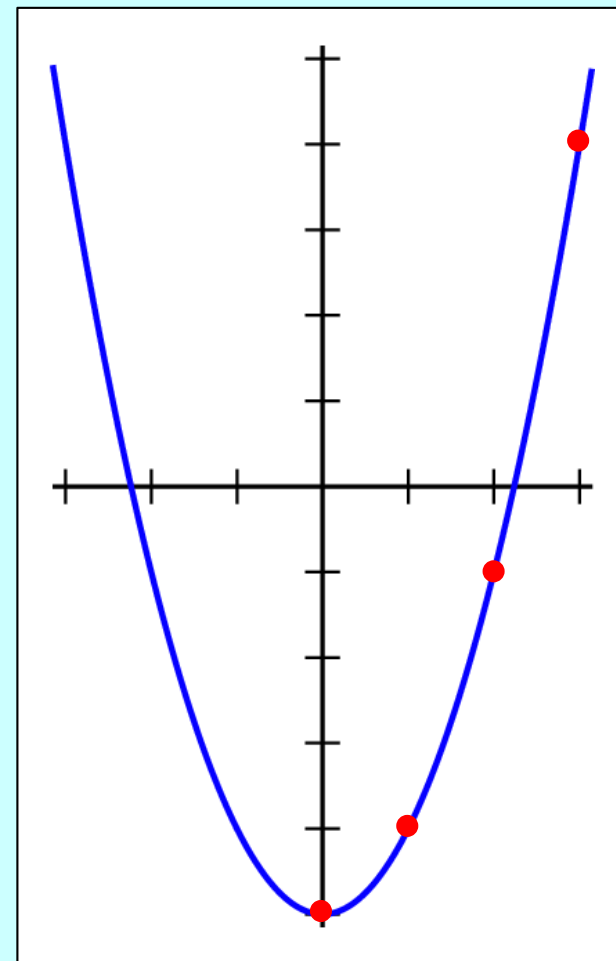
$$f(1) = 1^2 - 5 = -4$$

$$f(2) = 2^2 - 5 = -1$$

$$f(3) = 3^2 - 5 = 4$$

- The Python version of $f(x)$ is

```
def f(x):  
    return x ** 2 - 5
```



Built-In Functions

- To make programming easier, all modern languages (including Python, of course) include collections of predefined functions. The built-in functions that operate on numeric data appear in the following table:

abs (<i>x</i>)	The absolute value of <i>x</i>
max (<i>x</i> , <i>y</i> , ...)	The largest of the arguments
min (<i>x</i> , <i>y</i> , ...)	The smallest of the arguments
round (<i>x</i>)	The value of <i>x</i> rounded to the nearest integer
int (<i>x</i>)	The value of <i>x</i> truncated to an integer
float (<i>x</i>)	The value of <i>x</i> converted to floating-point

Importing Library Functions

- In addition to the built-in functions, Python offers a larger set of resources, which are organized into collections called *libraries*, or *modules*. Before you can use a library function, you must *import* that library in one of two ways.
- The most common strategy is to use an `import` statement to acquire access to that library's facilities without naming any specific functions. For example, the statement

```
import math
```

indicates that your program will use resources from the `math` library. When you do, you must use the *fully qualified name*, which includes the library name, as in `math.sqrt`.

- You can also use the `from-import` statement to gain access to specific functions without having to include the library name

```
from math import sqrt, max, sin, cos
```

Useful Entries in Python's **math** Library

<code>math.pi</code>	The mathematical constant π
<code>math.e</code>	The mathematical constant e
<code>math.sqrt(x)</code>	The square root of x
<code>math.floor(x)</code>	Rounds x down to the nearest integer
<code>math.ceil(x)</code>	Rounds x up to the nearest integer
<code>math.log(x)</code>	The natural logarithm of x
<code>math.log10(x)</code>	The common (base 10) logarithm of x
<code>math.exp(x)</code>	The inverse logarithm (e^x)
<code>math.sin(θ)</code>	The sine of θ , measured in radians
<code>math.cos(θ)</code>	The cosine of θ , measured in radians
<code>math.atan(x)</code>	The principal arctangent of x
<code>math.atan2(y, x)</code>	The arctangent of y / x
<code>math.degrees(θ)</code>	Converts from radians to degrees
<code>math.radians(θ)</code>	Converts from degrees to radians

Writing Your Own Functions

- The general form of a Python function definition is

```
def name (parameter list) :  
    statements in the function body
```

where *name* is the name of the function, and *parameter list* is a list of variables used to hold the values of each argument.

- You can return a value from a function by including a **return** statement, which is usually written as

```
return expression
```

Note the lack of curly braces and semicolons. Python instead relies on strict indentation to specify block structure.

Examples of Simple Functions

- The following function converts Fahrenheit temperatures to their Celsius equivalent:

```
def ftoc(f):  
    return 5 / 9 * (f - 32)
```

- The following function computes the volume of a cylinder of radius r and height h .

```
def cylinderVolume(r, h):  
    return math.pi * r ** 2 * h
```

Built-In Functions for Strings

- Python includes the following built-in functions to operate on string values:

len (<i>s</i>)	The number of characters in <i>s</i>
str (<i>x</i>)	The value of <i>x</i> converted to a string
int (<i>s</i>)	The string <i>s</i> interpreted as an integer
float (<i>s</i>)	The string <i>s</i> interpreted as floating-point
print (...)	Prints the arguments separated by spaces
input (<i>prompt</i>)	Reads a string from the user

- Python programs often require that user type in free form text in response to **input** calls. This underscores a key difference between JavaScript programs, which are almost always event-driven, and Python programs, which are often *scripts* that run uninterrupted, save for occasional calls to **input**.

Running Command-Line Programs

- Although it is possible to import programs into PyCharm and run them there, most Python programs are created as text files and then invoked from a command line.
- Python programs specify what happens when the program is run from the command line using the following lines at the end of the program file:

```
if __name__ == "__main__":  
    function ()
```

where *function* is the name of the function that serves as an entry point into the entire program.

- Lines of this sort are often called ***boilerplate***. It's sometimes better to just memorize the boilerplate than to understand it, and this is one of those times.

The AddTwoIntegers Program

```
# File: AddTwoIntegers.py

"""
This program adds two integers entered by the user.
"""

def AddTwoIntegers():
    print("This program adds two integers.")
    n1 = int(input("Enter n1? "))
    n2 = int(input("Enter n2? "))
    sum = n1 + n2
    print("The sum is", sum)

# Startup code
if __name__ == "__main__":
    AddTwoIntegers()
```

Boolean Expressions in Python

- Unsurprisingly, Python defines operators that work with Boolean data: *relational operators* and *logical operators*.
- There are six relational operators that compare values of other types and produce a **True/False** result (note **==** versus **===**):

==	Equals	!=	Not equals
<	Less than	<=	Less than or equal to
>	Greater than	>=	Greater than or equal to

Of course, the expression **n <= 10** evaluates to **True** if **n** is less than or equal to 10 and evaluates to **False** otherwise.

- Whereas JavaScript has **&&**, **||**, and **!**, Python has these:

and	Logical AND	<i>p and q</i> means both <i>p</i> and <i>q</i>
or	Logical OR	<i>p or q</i> means either <i>p</i> or <i>q</i> (or both)
not	Logical NOT	not <i>p</i> means the opposite of <i>p</i>

The **if** Statement

- The simplest of the control statements is the **if** statement, which occurs in two forms. You use the first when you need to perform an operation only if a particular condition is true:

if *condition*:
 statements to be executed if the condition is true

- You use the second form whenever you need to choose between two alternative paths, depending on whether the condition is true or false:

if *condition*:
 statements to be executed if the condition is true
else:
 statements to be executed if the condition is false

Functions Involving Control Statements

- The body of a function can contain statements of any type, including control statements. As an example, the following function uses an `if` statement to find the larger of two values:

```
def max(x, y):  
    if x > y:  
        return x  
    else:  
        return y
```

- Notice the reliance on the `:` and regimented indentation to specify block structure. It's very easy to identify Python code because of this.
- Python doesn't require semicolons to terminate expressions as our JavaScript programs do.

The **while** Statement

- The **while** statement is the simplest of Python's iterative control statements and has the following form:

```
while condition:  
    statements to be repeated
```

- A particularly useful **while** loop pattern we never needed in JavaScript looks like this:

```
while True:  
    line = input(prompt)  
    if line == "": break  
    rest of loop body
```

- The **while True** control line loops forever. The loop exits when the **break** statement is executed at the end of the input.

The AddList Program

```
# File: AddList.py

def AddList():
    print("This program adds a list of integers.")
    print("Enter a blank line to stop.")
    sum = 0
    while True:
        line = input(" ? ")
        if line == "": break
        sum += int(line)
    print("The sum is", sum)

# Startup code
if __name__ == "__main__":
    AddList()
```

The **for** Statement

- The **for** statement is Python's most powerful mechanism for specifying iteration and has the following general form:

```
for var in iterable:  
    statements to be repeated
```

- In this pattern, *var* is a variable name and *iterable* is any expression that produces a value that supports iteration.
- The effect of the **for** statement is to iterate over each value produced by the iterable object and assign it to *var*. The **for** loop continues as long as there are more values in the iterable.

The **range** Function

- The most common **for** loop pattern uses the built-in **range** function to produce the iterable object.
- The **range** function can take one, two, or three arguments, as follows:
 - range** (*limit*) counts from 0 up to *limit*−1
 - range** (*start*, *limit*) counts from *start* up to *limit*−1
 - range** (*start*, *limit*, *step*) counts by *step*
- Note that the **range** function always stops one step before the *limit* value is reached.
- The first of the three range structures above is the most common.

The `fact` Function

- The *factorial* of a number n (which is usually written as $n!$ in mathematics) is defined to be the product of the integers from 1 up to n . Thus, $5!$ is equal to 120, which is $1 \times 2 \times 3 \times 4 \times 5$.
- The following function definition uses a `for` loop to compute the factorial function:

```
def fact(n):  
    result = 1  
    for k in range(1, n + 1):  
        result *= k  
    return result
```

Iterating over Sequences

- The `for` statement is also used to iterate over the elements of a sequence of values.
- Python supports many different kinds of sequences that you will learn over the course of the next three weeks. The only sequence type you've seen so far is the string, which represents a sequence of characters.
- The following function returns the number of times the character `c` appears in the string `s`:

```
def countOccurrences(str, ch):  
    count = 0  
    for c in str:  
        if c == ch:  
            count += 1 # no ++ operator  
    return count
```

The End