Lecture 02: Data, Transformations, and Variables

Eric Franzosa, Ph.D.

franzosa@hsph.harvard.edu

Outline

- Elementary Python data types
- Functions, methods, and operators
- Variables

Aside: Comments

- In addition to data and transformations, computer code contains **comments**
- Comments are ignored by the computer processing your code
- They are intended to help future programmers understand the code, including future you
- Python includes two types of comments

In [ ]:
```
# I'm a single-line comment (starting with the '#' character)
```

In [ ]:
```
"""
I'm a multi-line comment,
also called a docstring,
enclosed in triple quotes
"""
```

Numbers and letters

- The "atoms" of the data we'll work with will usually be numbers and text (strings)
- Numbers come in two types: the `int` and `float`
- Strings are represented by the `str` type

`ints`

- The `int` type represents *integers*, i.e. counting numbers
- A surprisingly large number of problems boil down to counting
- Computers are very good at discrete math

In [ ]:
```
# the number 5
5
```

In [ ]:
```
# negative numbers work as you'd expect
-25
```

`float`s

- The `float` represents a decimal or "floating point" number
- These are more common in scientific computing
- Continuous math is harder for computers

In [ ]:

```
# a floating point number
3.1416
```

In [ ]:

```
# a floating point number in scientific notation
1e-5
```

`str`s

- The `str` represents a chunk of text, or string
- String data are always enclosed by quotes
  - This distinguishes them from other "words" in the code

In [ ]:

```
# strings be enclosed with double quotes
"banana"
```

In [ ]:

```
# or single quotes
'bananarama'
```

In [ ]:

```
# strings can contain multiple words and punctuation
"Hi Class, I'm a string with multiple words!"
```

In [ ]:

```
# you can use triple quotes to define multi-line strings
"""I am an
example of a multi-line
string"""
```

- Note that when Python evaluated the multi-line string, it saw the newline as a special character, `\n` (we'll talk more about these special characters later).

Transformations

- Transformations come in three major flavors
  - Operators
  - Functions
  - Methods

- Each transformation has several key properties
  - It takes some input data, called the argument(s)
  - It `return`s some output data
  - The `return`ed data can be futher transformed

Operators

- Operators are usually represented by symbols
  - (Sometimes short words)
- An operator's arguments are arranged around the symbol
- Some of these will be familiar to you...

In [ ]:

```
# the addition operator takes two numbers as input and returns their sum
1 + 2
```

In [ ]:

```
# the multiplication operators takes two numbers as input and returns their prod
uct
3.1416 * 3.1416
```

Other mathematical operators

In [ ]:
```
# exponentiation
3 ** 3
```

In [ ]:
```
# division
5 / 2
```

In [ ]:
```
# floor division
5 // 2
```

In [ ]:
```
# modulus (remainder division)
5 % 2
```

operators on strings

- The behavior of an operator depends on the `type` of its input data
- What happens if we try to "add" two string?

In [ ]:
```
"banana" + "rama"
```

- + concatenates a pair of strings

- Not every combination works
- We can't "add" a number and a string, even if the string looks like a number

In [ ]:
```
"five" + 5
```

In [ ]:
```
"5" + 5
```

Aside: Operators

- Operators aren't "smart"
- They rely on their input data to tell them what to do
- We'll learn more about this when we discuss object-oriented programming

- We *can* multiply a string by a number
- Try to guess what the result will be!

In [ ]:
```
2 * "banana"
```

In [ ]:
```
100 * "A"
```

functions

- A function is identified by a name
- The function is called by adding `()` after the name
- The function's arguments are provided ("passed") inside the `()`s

Anatomy of a function with one argument:

```
function_name( argument )
```

Anatomy of a function with multiple (3) arguments:

```
function_name( arg1, arg2, arg3 )
```

- Any Python code has access to a small number of "built-in" functions
- Some of them perform very intuitive functions on data

In [ ]:
```
# abs( ) returns the absolute value of its input
abs( -5.125 )
```

In [ ]:
```
# min returns the smallest of its inputs
min( 4, 2, -1, 7 )
```

In [ ]:
```
# max returns the largest
max( 4, 2, -1, 7 )
```

`print( )`

- `print( )` is a very important function
- It writes its arguments, usually strings, to the screen
- **Key Point**: This is how we monitor the state of a Python script
  - Jupyter shows us `return` values as `Out[]` blocks
  - In scripted code, only the computer sees these
- `print( )` makes data human-readable

In [ ]:
```
print( "I am an\nexample of a multi-line\nstring" )
```

- `print( )` does not `return` any data

functions for converting data types

In [ ]:
```
# str( ) returns its input as a string
str( 3.1416 )
```

In [ ]:
```
# int( ) returns its input as an int
int( "5" )
```

In [ ]:
```
int( 3.1416 )
```

In [ ]:
```
# float( ) returns its input as a float
float( 3 )
```

Read the docs!

- You can learn more about Python's built-in functions here:

  https://docs.python.org/3/library/functions.html
  (https://docs.python.org/3/library/functions.html)

- We'll encounter others moving forward.
- We'll also learn how to import additional functions for special tasks.

Methods

- Methods are functions that belong to data of a certain `type`.
- They tend to perform functions that are specifically relevant to that data.
- Methods can be identified by their `.` (dot) syntax.

Anatomy of a method call:

```
DATA.method_name( )
```

`DATA` above behaves like an argument to a non-method function:

```
function_name( DATA )
```

Methods can take extra arguments: `DATA.method_name( arg1, arg2, arg3 )`

String methods

- Strings have lots of methods useful for text manipulation
- Let's look at some to make this more concrete

In [ ]:
```
# .upper( ) is a string method that returns an upper-case version of the string
"bananarama".upper( )
```

In [ ]:
```
# .replace( X, Y ) returns a version of the string with Xs replaced by Ys
"bananarama".replace( "a", "o" )
```

Read the docs!

- Python data types (e.g. `str`) have associated docs:

  https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
  (https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str)

- These describe all of that type's special methods (among other things).
- Methods and functions form a "vocabulary" you'll build as you learn to program.

Key Idea: `return`ed data is data

- We can act on it
- We can use it as an operator argument
- We can use it as a function argument
- We can call methods on it

- This is intuitive for math operations
- Consider `1 + 2 + 3`
  - `1 + 2` returns `3`
  - then `3 + 3` returns `6`

In [ ]:
```
1 + 2 + 3
```

- Python follows an expanded "order of operations"
  - The math part will match your intuition

In [ ]:
```
1 + 2 * 3
```

- But this key idea is not specific to numbers/math

In [ ]:
```
"ba" + "na" + "na"
```

In [ ]:
```
"ba" + "na" * 2
```

- Nor is this key idea is not specific to operators

In [ ]:
```
int( "3" ) ** 3
```

- `int( "3" )` returns `3`
- then `3 ** 3` returns `27`

In [ ]:
```
abs( max( -7, -5, -3 ) )
```

- `max( -7, -5, -3 )` returns `-3`
- then `abs( -3 )` returns `3`

- It works with method calls as well
- The appearance can be less intuitive

In [ ]:
```
"bananarama".upper( ).replace( "A", "O" )
```

In [ ]:
```
"bananarama".replace( "A", "O" ).upper( )
```

- Why do the above-two method "chains" produce different final outputs?

Variables

- Variables are structures in programming for storing data
- Some store a constant value, but most change their contents as a program runs
- We **define** (and later **update**) variables with the = operator
    - In this context, we read = as "gets" rather than "equals"

In [ ]:

```
# define a variable called "my_number" with the value of 5
my_number = 5
```

- Unlike previous operators, = does not return data (no `Out[]` block)

- Variables are an **extremely** important concept in programming
- Why? Recall that programming is all about transformations of data
- Almost all data will be stored in variables

- Very little data is "hard-coded"
  - i.e. explicitly written out in the code
- Input data is usually read in from files
- Aside from final output data, most data will exist only in variables
  - i.e. never seen by the user

Variable naming

- Can't start with a number
- Otherwise a mix of letters, numbers, and _ as "spaces" in longer names
- Quotes `"`/`'` distinguish strings of text data from variable names

In [ ]:
```
# Variable names can be quite short (faster to type, harder to understand)...
a = 5
```

In [ ]:
```
# ...or quite long (harder to type, easier to understand)
bst273_instructor_lastname = "Franzosa"
```

In [ ]:
```
# lots of room for personal style
strCamelCaseVariable = "Hungarian Notation"
```

Key Concept: Acting on a variable means acting on its stored data

In [ ]:

```
# define some variables
number = 5
text = "Hello"
```

In [ ]:

```
number + number
```

In [ ]:

```
text + text
```

In [ ]:

```
text * number
```

- Redefining a variable will change the results of transformations we apply to it

In [ ]:

```
text = "Monkey"
text * number
```

In [ ]:

```
number = 3
text * number
```

- This holds for functions and methods as well

In [ ]:
```
text.upper( )
```

In [ ]:
```
text.upper( ).replace( "E", str( number ) )
```

Key Concept: Variables can capture returned values

In [ ]:
```
answer = 5 + 5
```

In [ ]:
```
print( answer )
```

In [ ]:
```
full_name = "Eric" + " " + "Franzosa"
```

In [ ]:
```
print( full_name )
```

Key Concept: Updating variables

- In Python, numbers and strings are immutable
- I.e. transforming number and string input data doesn't change the original data
- Rather, new output data is returned

In [ ]:
```
number = 5
number + 1
```

In [ ]:
```
print( number )
```

- Update a variable by redefining it (using the = operator)

In [ ]:
```
number = 5
number = number + 1
print( number )
```

Aside: Order of execution

- Code tends to run from top to bottom
- This is especially true with a Python script
- This means we get into trouble if we try to use a variable before defining it

In [ ]:
```
print( cool_variable ** 2 )
cool_variable = 7
```

- Things get a little weird in Jupyter Notebooks since we can execute code in any order we like
- Why does the following work?

In [ ]:
```
print( number ** 2 )
number = 5
```

Aside: Coding Style

- Some aspects of coding are rigid
  - i.e. Things will break if you don't do them the single correct way
- Others are flexible
  - e.g. white space around arguments
  - I prefer the extra space for readability

In [ ]:
```
# operator without white space
1+2
```

In [ ]:
```
# operator with white space (padding)
1 + 2
```

In [ ]:
```
# function call without white space
min(1,2,3,4)
```

In [ ]:
```
# function call with white space (padding)
min( 1, 2, 3, 4 )
```

Practice: `sums` (easy)

In [ ]:
```
# (1) write code to sum the numbers 3 through 7 using the "+" operator
```

In [ ]:
```
# (2) write code to sum the numbers 3 through 7 using a single function call
```

- The sum of the numbers from 1 to N is equal to N * (N + 1) / 2

In [ ]:
```
# (3) write code to use the formula above to compute the sum of the numbers 1 th
rough 100
```

Practice: `str`ing manipulation (medium)

In [ ]:
```
# (0) Don't forget to execute this cell to define these variables.
sa = "ba"
sb = "na"
```

In [ ]:
```
# (1) Return the string "banana" by transforming <sa> and <sb> above.
```

In [ ]:
```
# (2) Reuse your code in (1) to return the string "bonono" in by redefining <sa>
 and <sb> above.
```

In [ ]:
```
# (3) Return the string "banana" by adding two method calls to the string below.
"beNeNe"
```

Practice: Round things (medium)

- `round( X, Y )` is a built-in function
- It returns the input number X rounded to Y (an `int`) decimal places

In [ ]:
```
# (0) Don't forget to execute this cell to define <pi>
pi = "3.1416"
```

In [ ]:
```
# (1) Without typing any digits, convert (update) <pi> to hold a decimal number.
```

In [ ]:
```
# (2) Using the round( ) function, update <pi> to be rounded to TWO decimal plac
es.
```

In [ ]:
```
# (3) Tau is a constant equal to two times pi. Define a <tau> variable below.
```

In [ ]:
```
# (4) Use <tau> to compute the circumference of a circle with radius r = 2.
r = 2
area = 999
print( area )
```

Practice: Advanced updates (hard)

In [ ]:
```
# (0) Don't forget to execute this cell to define these variables.
S1 = 2
S2 = 3
```

In [ ]:
```
# (1) Write code to use the variable <S3> to exchange the values of <S1> and <S2
>.
S3 = 999
print( S1, S2 )
```

- Consider a sequence S(1), S(2), S(3), ... , S(n) where S(1) and S(2) are known and S(n) = S(n-1) + S(n-2)
    - For example, 1, 1, 2, 3, 5, 8, etc.
- For any S(1) and S(2), the ratio of S(n+1) to S(n) always approaches the **golden ratio** ~ 1.613

In [ ]:
```
# (2) Add a print statement below to show that S3 / S2 approaches 1.613 as this
 block is executed repeatedly.
S3 = S1 + S2
S1 = S2
S2 = S3
```

In [ ]:
```
# (3) Redefine S1 and S2 above, then re-assess if the claim in (2) holds.
```