Lecture 03 - Variables, Scalars and Functions

2018-09-11 Kevin Bonham, PhD

Outline

- Meaning vs syntax
- Variables and methods
- `Type`s of information (scalars)
- **Hands-on**: working with `Boolean`s and `String`s

Learning Ojectives

At the end of this class period, you should be able to:

1. Identify and explain the difference between common scalar types
2. Create and modify variables in python
3. Perform basic arithmetic calculations in code
4. Manipulate strings and print the results to the console

Meaning vs Syntax

- A programming "language" is really a translation
  - Human intent --> machine code
- There are two basic features in any language:
  - Data (information)
  - Instructions (actions)

The type of translation between intent and machine code is the ”syntax”

For example, to ask the computer to display some text on the screen...

- In python 2: `print "Hello, World!"`
- In python 3: `print("Hello, World!")`
- In julia: `println("Hello, World!")`
- In java: `System.out.println("Hello, World!")`

We will be using python3 syntax

But many (most?) of the concepts you learn in this course are applicable to other languages too. Just append `in {LANGUAGE}` to your google search to learn the syntax. Eg:

In [28]:
```
for language in ["python 2", "python 3", "julia", "java", "ruby", "perl"]:
    print("How do I print something to the screen in {}?".format(language))
```

```
How do I print something to the screen in python 2?
How do I print something to the screen in python 3?
How do I print something to the screen in julia?
How do I print something to the screen in java?
How do I print something to the screen in ruby?
How do I print something to the screen in perl?
```

Exercise

Find the syntax for exponentiation in python

In this course, we will not always provide the answer ahead of time. Many solutions are just a Google (or Bing) search away.

In [ ]:
```
# make this cell return 2 raised to the eighth power

2 8
```

Variables store information

It is often useful to pass around data with names, rather than passing around the values themselves.

In [29]:
```
2 + 2 # this value isn't stored anywhere
```

Out[29]:
```
4
```

But we can "assign" it to a variable (in this case, `x`) with =:

In [30]:
```
x = 2 + 2
```

Now we can access the value in the variable:

In [31]:
```
x
```

Out[31]: 4

In [32]:
```
print("The value of x is", x)
```

```
The value of x is 4
```

And perform operations with it:

In [33]:
```
x * 4
```

Out[33]:
```
16
```

Variables behave exactly like their values

In other words,

In [34]:
```
print("The value of x is", 4)
```

```
The value of x is 4
```

is identical to

In [35]:
```
x = 4
print("The value of x is", x)
```

```
The value of x is 4
```

Variables can be reassigned

In [36]:
```
counter = 1
counter
```

Out[36]: 1

In [37]:
```
counter = counter + 1 # can also write `counter += 1`
counter
```

Out[37]: 2

In [38]:
```
counter = 1

# This line says: perform the following action 10 times
for i in range(10):
    # This line says, add 1 to the value of counter
    counter = counter + 1
```

Poll: Week 1, lecture 3

Question 1

What is the value of the `counter` variable?

1. 1
2. 2
3. 10
4. 11

In [ ]:
```
counter
```

Pay attention to the difference between the following:

In [39]:
```
# This is identical to the previous version we saw
counter = 1
for i in range(10):
    counter = counter + 1

# What's different here?
counter_2 = 1
for i in range(10):
    counter_2 + 1
```

Poll: Week 1, lecture 3

Question 2

What is the value of the variable `counter_2`?

1. 1
2. 2
3. 10
4. 11

"Functions" perform actions on data

You've already seen a couple of functions in action eg:

In [40]:
```
print("Hello, World!")

for i in range(10):
    counter = counter + 1
```

```
Hello, World!
```

Functions may also ”return” results

In [41]:
```
sum([1,2,3,4])
```

Out[41]:
```
10
```

In [42]:
```
print("Hi, I'm Z!")
```

```
Hi, I'm Z!
```

When we assign variables with functions we assign the "return" value of the function

In [43]:
```
y = sum([1,2,3,4])
z = print("Hi, I'm Z!")
```

```
Hi, I'm Z!
```

In [ ]:
```
print(y)
print(z)
```

Functions take 0 or more "arguments"

In [ ]:
```
s1 = "Hello,"
s2 = "World!"

print(s1)
print(s2)
print()
print(s1, s2) # arguments are separated by commas
```

When writing functions, arguments (`args`) are like variables

In [ ]:
```
def weird_addition(number1, number2):
    result = 2 * (number1 + number2)
    return result

weird_addition(3, 10)
```

Complete the function

Complete the following function to divide the first argment by the square of the second argument and return the result.

When this code block is evaluated, it should print "42.0".

In [ ]:
```
def square_divide(): # What should the arguments for this function be?
    sd = n1 / n2 ** 2 # In python, `**` is the syntax for exponents
    # don't forget to return the result


result = square_divide(168, 2)

print(result)
```

`Types` of information (scalars)

- Different sorts of information are stored differently by computers
- Most programming languages have built-in data types
  - Most also have the ability to create your own
- Common simple data types are:
  - Strings: sequences of characters surrounded by quotes: `"Hello, World!"`
  - Integers: positive or negative whole numbers: `42`
  - Floats: numbers with decimals: `3.142`
  - Booleans: binary values, `True` or `False`

The actions of functions depend on their arguments

In [ ]:
```
"Hello, " + "World!"
```

In [ ]:
```
20 + 22
```

In [ ]:
```
20 + 22.0
```

In [ ]:
```
20 + "22.0"
```

"Scalars" are single values

- Ints, Floats, and Bools are examples of scalars
- There are also containers for holding multiple objects (Thursday)
- Strings are weird
  - Can be thought of as scalars
  - Can be thought of as array of characters

A final note on python functions

- Many python types have internal "methods"
- Methods are another name for functions that act on a type
- In python, internal methods are called with dot syntax

In [ ]:
```
my_string = "I am a happy string, all held together"
my_string.split()
```

Internal methods may also take arguments

In [ ]:
```
my_other_string = "Please! Dont pull me apart! I cant take it!"
my_other_string.split("!")
```

Hands on practice

- Download the `03-variables-scalars-functions.ipynb` file from Canvas
- Launch with `jupyter notebook`
- Execute the cells in order (`shift+enter` evaluates a cell)
- Cells preceded by a **bold** question may require some editing to the the correct output
  - ***expected output will be in bold italics***
- Try to fix errors or correct the code to get the right output.

Googling is permitted, encouraged even! Collaboration is also ok, but make sure you understand what's happening.

Working with logic / booleans

Boolean values are binary values: `True` or `False`. These values are returned when we ask questions about equality or compare values to see if one is less than or greater than the other

In python, the syntax for this is == for "is equal?", `!`= for "is not equal?", > and < for "is greater than?" and "is less than?", and >= and <= for "is greater/less than or equal to?"

For example (see if you can predict the outcome before you evaluate the cell):

In [ ]:
```
1 == 1.0
```

In [ ]:
```
1 == 2
```

In [ ]:
```
1 != 2
```

In [ ]:
```
100 > 10
```

In [ ]:
```
100 < 10 * 10
```

In [ ]:
```
100 <= 10 * 10
```

We can also do boolean logic (https://en.wikipedia.org/wiki/Boolean_algebra#Operations)

In [ ]:
```
True and False
```

In [ ]:
```
x = 1
y = 10

y < x or x * 10 == y
```

In [ ]:
```
True and not False
```

In each of the following, change only one boolean value to get the cell to return `True`

In [ ]:
```
False or False # eg, make one of these True
```

In [ ]:
```
False and True
```

In [ ]:
```
# pay attention to order of operations
False and True or False
```

In [ ]:
```
False or True and False
```

In [ ]:
```
False or (True and False)
```

In each of the following, change only one of the following:

- == (is equal?) to `!=` (is not equal?) or vice versa
- `and` to `or` or vice versa
- insert or remove a `not`

to get the cell to evaluate to `True`

NOTE: In most cases there are multiple correct answers. Try out different methods.

In [ ]:
```
e = 2.72
pi = 3.14
lue = 42 # Life, the Universe and Everything

e < pi and not pi > lue
```

In [ ]:
```
e == pi or e == lue
```

In [ ]:
```
False and e == pi or e == lue
```

In [ ]:
```
(False and False or False) and True
```

In [ ]:
```
"What makes strings less than?" < "I bet I know..."
```

String practice

Strings are sequences of characters. You can concatenate (merge) strings by adding them together.

In [ ]:
```
"One String. " + "Two String. " + "Red String. " + "Blue String."
```

Some arithmetic operations also work:

In [ ]:
```
"Why? Because. " * 10
```

Notice that the spaces have to be included when concatenating:

In [ ]:
```
a = "No"
b = "bueno"
a + b
```

When printing, spaces are added between arguments

In [ ]:
```
print(a, b)
```

The following strings will be used in some of the following exercises. Evaluate the cell, but do not change it.

In [ ]:
```
s1 = "The short"
s2 = "brown fox"
s3 = "jumped over"
s4 = "the lazy dog"
s5 = "ban"
s6 = "ana"
s7 = "na"
```

Get the output of each of the following cells to match the text in *italics* above it. Try adding the least amount of text possible.

"The short brown fox jumped over the lazy dog"

In [ ]:
```
print(s1, s4)
```

Same thing, but you can't add commas

In [ ]:
```
print(s1 + s2, s3, s4)
```

bananananananana

In [ ]:
```
print(s5 + s6 + s7) # you can do this with only 2 characters
```

Recall that the `.split()` function acts on a string to divide it at spaces (if no arguments are passed), or at a character if a character is passed as an argument.

In [ ]:
```
a = "Whoa... what happened?".split()
a
```

The `.join()` function does the inverse. It's a method of a string (the separator) and takes a list as an argument (we'll learn more about lists on Thursday, it's what's returned by

```
.split())
```

In [ ]:
```
" ".join(a)
```

Get the following to return *The-short-brown-fox-jumped-over-the-lazy-dog*

In [ ]:
```
my_string = s1 + ' ' + s2 + ' ' + s3 + ' ' + s4

my_split = my_string.split()

join(my_split)
```