lab01

August 27, 2024

[48]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab01.ipynb")
```

1 Lab 1: Expressions

Welcome to Data Science for All!

Please refer to canvas for course policies and instructions on how to submit assignments.

Today's lab In today's lab, you'll learn how to:

1. navigate Jupyter notebooks (like this one);
2. write and evaluate some basic *expressions* in Python, the computer language of the course;
3. call *functions* to use code other people have written; and
4. break down Python code into smaller parts to understand it.

This lab covers parts of Chapter 3 of the online textbook. You should read the examples in the book, but not right now.

2 Lab Warm Up!

Every lab will start with a class warm up activity. We will work together as a class in the following coding cells to prepare you for all sections of this lab.

Make sure to come to lab on time so you don't miss points for this warm-up!

Since this is lab01, our warm up will be starting the lab together as a group.

Let's get started!

3 1. Jupyter notebooks

This webpage is called a Jupyter notebook. A notebook is a place to write programs and view their results, and also to write text.

3.1 1.1. Text cells

In a notebook, each rectangle containing text or code is called a *cell*.

Text cells (like this one) can be edited by double-clicking on them. They're written in a simple format called Markdown to add formatting and section headings. You don't need to learn Markdown, but you might want to.

After you edit a text cell, click the "run cell" button at the top that looks like | or hold down `shift` + `return` to confirm any changes. (Try not to delete the instructions of the lab.)

Question 1.1.1. This paragraph is in its own text cell. Try editing it so that this sentence is the last sentence in the paragraph, and then click the "run cell" | button or hold down `shift` + `return`.

3.2 1.2. Code cells

Other cells contain code in the Python 3 language. Running a code cell will execute all of the code it contains.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, either press | or hold down `shift` + `return`.

Try running this cell:

[49]:
```
print("Hello, World!")
```

```
Hello, World!
```

And this one:

[50]:
```
print("\N{WAVING HAND SIGN}, \N{EARTH GLOBE ASIA-AUSTRALIA}!")
```

```
 ,  !
```

The fundamental building block of Python code is an expression. Cells can contain multiple lines with multiple expressions. When you run a cell, the lines of code are executed in the order in which they appear. Every `print` expression prints a line. Run the next cell and notice the order of the output.

[51]:
```
print("First this line,")
print("then the whole \N{EARTH GLOBE ASIA-AUSTRALIA},")
print("and then this one.")
```

```
First this line,
then the whole  ,
and then this one.
```

Question 1.2.1. Using the first coding cells as examples, change the cell above so that it prints out:

```
First this line,
then the whole  ,
and then this one.
```

Hint: If you're stuck on the Earth symbol for more than a few minutes, try talking to a neighbor or a staff member. That's a good idea for any lab problem.

3.3 1.3. Writing Jupyter notebooks

You can use Jupyter notebooks for your own projects or documents. When you make your own notebook, you'll need to create your own cells for text and code.

To add a cell, click the + button in the menu bar. It'll start out as a text cell. You can change it to a code cell by clicking inside it so it's highlighted, clicking the drop-down box next to the restart ( ) button in the menu bar, and choosing "Code".

Question 1.3.1. Add a code cell below this one. Write code in it that prints out:

```
A whole new cell! ♪♪
```

(That musical note symbol is like the Earth symbol. Its long-form name is `\N{EIGHTH NOTE}`.)

Run your cell to verify that it works.

```
[52]: print("A whole new cell! \N{EIGHTH NOTE}\N{EARTH GLOBE ASIA-AUSTRALIA}\N{EIGHTH␣
      ↪NOTE}")
```

```
A whole new cell! ♪♪
```

3.4 1.4. Errors

Python is a language, and like natural human languages, it has rules. It differs from natural language in two important ways: 1. The rules are *simple*. You can learn most of them in a few weeks and gain reasonable proficiency with the language in a semester. 2. The rules are *rigid*. If you're proficient in a natural language, you can understand a non-proficient speaker, glossing over small mistakes. A computer running Python code is not smart enough to do that.

Whenever you write code, you'll make mistakes. When you run a code cell that has errors, Python will sometimes produce error messages to tell you what you did wrong.

Errors are okay; even experienced programmers make many errors. When you make an error, you just have to find the source of the problem, fix it, and move on.

We have made an error in the next cell. Run it and see what happens.

```
[53]: print("This line is missing something.")
```

```
This line is missing something.
```

Note: In the toolbar, there is the option to click `Cell > Run All`, which will run all the code cells in this notebook in order. However, the notebook stops running code cells if it hits an error, like the one in the cell above.

You should see something like this (minus our annotations):

The last line of the error output attempts to tell you what went wrong. The *syntax* of a language is its structure, and this `SyntaxError` tells you that you have created an illegal structure. "`EOF`" means "end of file," so the message is saying Python expected you to write something more (in this case, a right parenthesis) before finishing the cell.

There's a lot of terminology in programming languages, but you don't need to know it all in order to program effectively. If you see a cryptic message like this, you can often get by without deciphering it. (Of course, if you're frustrated, ask a neighbor or a staff member for help.)

Try to fix the code above so that you can run the cell and see the intended message instead of an error.

3.5 1.5. The Kernel

The kernel is a program that executes the code inside your notebook and outputs the results. In the top right of your window, you can see a circle that indicates the status of your kernel. If the circle is empty ( ), the kernel is idle and ready to execute code. If the circle is filled in ( ), the kernel is busy running some code.

Next to every code cell, you'll see some text that says `In [...]`. Before you run the cell, you'll see `In [ ]`. When the cell is running, you'll see `In [*]`. If you see an asterisk (*) next to a cell that doesn't go away, it's likely that the code inside the cell is taking too long to run, and it might be a good time to interrupt the kernel (discussed below). When a cell is finished running, you'll see a number inside the brackets, like so: `In [1]`. The number corresponds to the order in which you run the cells; so, the first cell you run will show a 1 when it's finished running, the second will show a 2, and so on.

You may run into problems where your kernel is stuck for an excessive amount of time, your notebook is very slow and unresponsive, or your kernel loses its connection. If this happens, try the following steps: 1. At the top of your screen, click **Kernel**, then **Interrupt**. 2. If that doesn't help, click **Kernel**, then **Restart**. If you do this, you will have to run your code cells from the start of your notebook up until where you paused your work. 3. If that doesn't help, restart your server. First, save your work by clicking **File** at the top left of your screen, then **Save and Checkpoint**. Next, click **Control Panel** at the top right. Choose **Stop My Server** to shut it down, then **Start My Server** to start it back up. Then, navigate back to the notebook you were working on. You'll still have to run your code cells again.

4 2. Numbers

Quantitative information arises everywhere in data science. In addition to representing commands to print out lines, expressions can represent numbers and methods of combining numbers. The expression `3.2500` evaluates to the number 3.25. (Run the cell and see.)

[75]:
```
3.2500
```

[75]:
```
3.25
```

Notice that we didn't have to `print`. When you run a notebook cell, if the last line has a value, then Jupyter helpfully prints out that value for you. However, it won't print out prior lines automatically.

[76]:
```
print(2)
3
4
```

```
2
```

[76]:
```
4
```

Above, you should see that 4 is the value of the last expression, 2 is printed, but 3 is lost forever because it was neither printed nor last.

You don't want to print everything all the time anyway. But if you feel sorry for 3, change the cell above to print it.

4.1 2.1. Arithmetic

The line in the next cell subtracts. Its value is what you'd expect. Run it.

[77]:
```
3.25 - 1.5
```

[77]:
```
1.75
```

Many basic arithmetic operations are built into Python. The textbook section on Expressions describes all the arithmetic operators used in the course. The common operator that differs from typical math notation is `**`, which raises one number to the power of the other. So, `2**3` stands for 2^3 and evaluates to 8.

The order of operations is the same as what you learned in elementary school, and Python also has parentheses. For example, compare the outputs of the cells below. The second cell uses parentheses. Compare the two outputs.

[78]:
```
4+6*5-6*3**2*2**3/4*7
```

[78]:
```
-722.0
```

[80]:
```
4+(6*5-(6*3))**2*((2**3)/4*7)
```

[80]:
```
2020.0
```

In standard math notation, the first expression is

$$4 + 6 \times 5 - 6 \times 3^2 \times \frac{2^3}{4} \times 7,$$

while the second expression is

$$4 + (6 \times 5 - (6 \times 3))^2 \times (\frac{(2^3)}{4} \times 7).$$

Question 2.1.1. Write a Python expression in this next cell that's equal to $5 \times (3 + 11) - \frac{1}{3} + 4^{0.5} - \frac{8}{3} + 3$. That's five times three plus eleven, minus one third, plus two to the power of one half, minus seven thirty-thirds plus three.

[81]:
```
# Replace the ellipses (`...`) with your expression.  Try to use parentheses␣
↪only when necessary.
```

```
# HINT: The correct output should equal 72.0, so keep thinking about order of␣
↪operations if you get a different result.
5*(3+11)-1/3+4**0.5-8/3+3
```

```
[81]: 72.0
```

5 3. Names

In natural language, we have terminology that lets us quickly reference very complicated concepts. We don't say, "That's a large mammal with brown fur and sharp teeth!" Instead, we just say, "Bear!"

In Python, we do this with *assignment statements*. An assignment statement has a name on the left side of an `=` sign and an expression to be evaluated on the right.

```
[85]: ten = 3 * 2 + 4
```

When you run that cell, Python first computes the value of the expression on the right-hand side, `3 * 2 + 4`, which is the number 10. Then it assigns that value to the name `ten`. At that point, the code in the cell is done running.

After you run that cell, the value 10 is bound to the name `ten`:

```
[83]: ten
```

```
[83]: 10
```

The statement `ten = 3 * 2 + 4` is not asserting that `ten` is already equal to `3 * 2 + 4`, as we might expect by analogy with math notation. Rather, that line of code changes what `ten` means; it now refers to the value 10, whereas before it meant nothing at all.

If the designers of Python had been ruthlessly pedantic, they might have made us write

```
define the name ten to hereafter have the value of 3 * 2 + 4
```

instead.

Question 3.1. In the cell below, replace the ellipsis with code that uses a name (like type in `eleven` and run the cell) that hasn't been assigned to anything.

IMPORTANT: You should see an error because the name `eleven` does not refer to anything! That is the point of this exercise.

```
[86]: eleven
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[86], line 1
----> 1 eleven
```

```
NameError: name 'eleven' is not defined
```

A common pattern in Jupyter notebooks is to assign a value to a name and then immediately evaluate the name in the last line in the cell so that the value is displayed as output.

```
[87]: close_to_pi = 355/113
      close_to_pi
```

```
[87]: 3.1415929203539825
```

Another common pattern is that a series of lines in a single cell will build up a complex computation in stages, naming the intermediate results.

```
[89]: semimonthly_salary = 841.25
      monthly_salary = 2 * semimonthly_salary
      number_of_months_in_a_year = 12
      yearly_salary = number_of_months_in_a_year * monthly_salary
      yearly_salary
```

```
[89]: 20190.0
```

Names in Python can have letters (upper- and lower-case letters are both okay and count as different letters), underscores, and numbers. The first character can't be a number (otherwise a name might look like a number). And names can't contain spaces, since spaces are used to separate pieces of code from each other.

Other than those rules, what you name something doesn't matter *to Python*. For example, this cell does the same thing as the above cell, except everything has a different name:

```
[90]: a = 841.25
      b = 2 * a
      c = 12
      d = c * b
      d
```

```
[90]: 20190.0
```

However, names are very important for making your code *readable* to yourself and others. The cell above is shorter, but it's totally useless without an explanation of what it does.

5.1 3.1. Checking your code

Now that you know how to name things, you can start using the **built-in tests** to check whether your work is correct. Sometimes, there are multiple tests for a single question, and passing all of them is required to receive credit for the question. Please don't change the contents of the test cells.

Go ahead and attempt Question 3.1.2. Running the cell directly after it will test whether you have assigned `seconds_in_a_decade` correctly in Question 3.1.2. If you haven't, this test will tell you the correct answer. Resist the urge to just copy it, and instead try to adjust your expression. (Sometimes the tests will give hints about what went wrong…)

Question 3.1.2. Assign the name `seconds_in_a_decade` to the number of seconds between midnight January 1, 2010 and midnight January 1, 2020. Note that there are two leap years in this span of a decade. A non-leap year has 365 days and a leap year has 366 days.

Hint: If you're stuck, the next section shows you how to get hints.

[98]:
```
# Change the next line, adding additional lines if you'd like,
# so that it computes the number of seconds in a decade
# and assigns that number the name, seconds_in_a_decade.
# For full credit, be sure to show work like the yearly_salary
# example above.

# the number of days in a non-leap year
days_in_non_leap_year = 365

# the number of days in a leap year
days_in_leap_year = 366

# the number of seconds in a day
seconds_in_a_day = 24 * 60 * 60

# the number of non-leap years in the decade
number_of_non_leap_years = 8

# the number of leap years in the decade
number_of_leap_years = 2

# the number of seconds in all non-leap years
seconds_in_non_leap_years = number_of_non_leap_years * days_in_non_leap_year *␣
 ↪seconds_in_a_day

#the number of seconds in all leap years
seconds_in_leap_years = number_of_leap_years * days_in_leap_year *␣
 ↪seconds_in_a_day

# the total number of seconds in the decade
seconds_in_a_decade = seconds_in_non_leap_years + seconds_in_leap_years


# We've put this line in this cell
# so that it will print the value you've given to seconds_in_a_decade when you␣
 ↪run it.
# You don't need to change this.
seconds_in_a_decade
```

```
[98]: 315532800
```

```
[99]: grader.check("q3_1_2")
```

```
[99]: q3_1_2 results: All test cases passed!
```

6 3.2. Comments

You may have noticed these lines in the cell in which you answered Question 3.2:

```
# Change the next line, adding additional lines if you'd like,
# so that it computes the number of seconds in a decade
# and assigns that number the name, seconds_in_a_decade.
# For full credit, be sure to show work like the yearly_salary
# example above.
```

This is called a *comment*. It doesn't make anything happen in Python; Python ignores anything on a line after a `#`. Instead, it's there to communicate something about the code to you, the human reader. Comments are extremely useful.

Source: http://imgs.xkcd.com/comics/future_self.png

6.1 4. Calling functions

The most common way to combine or manipulate values in Python is by calling functions. Python comes with many built-in functions that perform common operations.

For example, the `abs` function takes a single number as its argument and returns the absolute value of that number. Run the next two cells and see if you understand the output.

```
[91]: abs(5)
```

```
[91]: 5
```

```
[92]: abs(-5)
```

```
[92]: 5
```

6.2 4.1. Application: Computing walking distances

Chunhua is on the corner of 7th Avenue and 42nd Street in Midtown Manhattan, and she wants to know far she'd have to walk to get to Gramercy School on the corner of 10th Avenue and 34th Street.

She can't cut across blocks diagonally, since there are buildings in the way. She has to walk along the sidewalks. Using the map below, she sees she'd have to walk 3 avenues (long blocks) and 8 streets (short blocks). In terms of the given numbers, she computed 3 as the difference between 7 and 10, *in absolute value*, and 8 similarly.

Chunhua also knows that blocks in Manhattan are all about 80m by 274m (avenues are farther apart than streets). So in total, she'd have to walk $(80 \times |42 - 34| + 274 \times |7 - 10|)$ meters to get to the park.

Question 4.1.1. Fill in the line `num_avenues_away = ...` in the next cell so that the cell calculates the distance Chunhua must walk and gives it the name `manhattan_distance`. Everything else has been filled in for you. **Use the `abs` function.** Also, be sure to run the test cell afterward to test your code.

[100]:
```
# Here's the number of streets away:
num_streets_away = abs(42-34)

# Compute the number of avenues away in a similar way:
num_avenues_away = abs(7-10)

street_length_m = 80
avenue_length_m = 274

# Now we compute the total distance Chunhua must walk.
manhattan_distance = street_length_m*num_streets_away +␣
 ↪avenue_length_m*num_avenues_away

# We've included this line so that you see the distance you've computed
# when you run this cell.
# You don't need to change it, but you can if you want.
manhattan_distance
```

[100]:
```
1462
```

[101]:
```
grader.check("q4_1_1")
```

[101]:
```
q4_1_1 results: All test cases passed!
```

Multiple arguments Some functions take multiple arguments, separated by commas. For example, the built-in `max` function returns the maximum argument passed to it.

[102]:
```
max(2, -3, 4, -5)
```

[102]:
```
4
```

7 5. Understanding nested expressions

Function calls and arithmetic expressions can themselves contain expressions. You saw an example in the last question:

```
abs(42-34)
```

has 2 number expressions in a subtraction expression in a function call expression. And you probably wrote something like `abs(7-10)` to compute `num_avenues_away`.

Nested expressions can turn into complicated-looking code. However, the way in which complicated expressions break down is very regular.

Suppose we are interested in heights that are very unusual. We'll say that a height is unusual to the extent that it's far away on the number line from the average human height. An estimate of the average adult human height (averaging, we hope, over all humans on Earth today) is 1.688 meters.

So if Kayla is 1.21 meters tall, then her height is $|1.21 - 1.688|$, or .478, meters away from the average. Here's a picture of that:

And here's how we'd write that in one line of Python code:

```
[103]: abs(1.21 - 1.688)
```

```
[103]: 0.478
```

What's going on here? `abs` takes just one argument, so the stuff inside the parentheses is all part of that *single argument*. Specifically, the argument is the value of the expression `1.21 - 1.688`. The value of that expression is `-.478`. That value is the argument to `abs`. The absolute value of that is `.478`, so `.478` is the value of the full expression `abs(1.21 - 1.688)`.

Picture simplifying the expression in several steps:

1. `abs(1.21 - 1.688)`
2. `abs(-.478)`
3. `.478`

In fact, that's basically what Python does to compute the value of the expression.

Question 5.1. Say that Paola's height is 1.76 meters. In the next cell, use `abs` to compute the absolute value of the difference between Paola's height and the average human height. Give that value the name `paola_distance_from_average_m`.

```
[104]: # Replace the ... with an expression
       # to compute the absolute value
       # of the difference between Paola's height (1.76m) and the average human height␣
        ↪(1.688m).
       paola_distance_from_average_m = abs(1.76-1.688)


       # Again, we've written this here
       # so that the distance you compute will get printed
       # when you run this cell.
       paola_distance_from_average_m
```

```
[104]: 0.07200000000000006
```

```
[105]: grader.check("q51")
```

[105]: q51 results: All test cases passed!

7.1 5.1. More nesting

Now say that we want to compute the more unusual of the two heights. We'll use the function `max`, which (again) takes two numbers as arguments and returns the larger of the two arguments. Combining that with the `abs` function, we can compute the larger distance from average among the two heights:

[106]:
```
# Just read and run this cell.

kayla_height_m = 1.21
paola_height_m = 1.76
average_adult_height_m = 1.688

# The larger distance from the average human height, among the two heights:   ␣
 ↪                                                                             ␣
 ↪
larger_distance_m = max(abs(kayla_height_m - average_adult_height_m),␣
 ↪abs(paola_height_m - average_adult_height_m))

# Print out our results in a nice readable format:
print("The larger distance from the average height among these two people is",␣
 ↪larger_distance_m, "meters.")
```

```
The larger distance from the average height among these two people is 0.478
meters.
```

The line where `larger_distance_m` is computed looks complicated, but we can break it down into simpler components just like we did before.

The basic recipe is to repeatedly simplify small parts of the expression: * **Basic expressions:** Start with expressions whose values we know, like names or numbers. - Examples: `paola_height_m` or `5`. * **Find the next simplest group of expressions:** Look for basic expressions that are directly connected to each other. This can be by arithmetic or as arguments to a function call. - Example: `kayla_height_m - average_adult_height_m`. * **Evaluate that group:** Evaluate the arithmetic expression or function call. Use the value computed to replace the group of expressions.
- Example: `kayla_height_m - average_adult_height_m` becomes `-.478`. * **Repeat:** Continue this process, using the value of the previously-evaluated expression as a new basic expression. Stop when we've evaluated the entire expression. - Example: `abs(-.478)` becomes `.478`, and `max(.478, .072)` becomes `.478`.

You can run the next cell to see a slideshow of that process.

[107]:
```
from IPython.display import IFrame
IFrame('https://docs.google.com/presentation/d/e/
 ↪2PACX-1vTiIUOa9tP4pHPesrI8p2TCp8WCOJtTb3usOacQFPfkEfvQMmX-JYEW3OnBoTmQEJWAHdBP6Mvp053G/
 ↪embed?start=false&loop=false&delayms=3000', 800, 600)
```

[107]: <IPython.lib.display.IFrame at 0x798aec2ed4e0>

Question 5.1.1. YOUR TURN! Given the heights of players from the Golden State Warriors, write an expression that computes the smallest difference between any of the three heights. Your expression shouldn't have any numbers in it, only function calls and the names `klay`, `steph`, and `dangelo`. Give the value of your expression the name `min_height_difference`.

[116]:
```
# HINT: Refer to the steps in the height example above if you get stuck
# The three players' heights, in meters:
klay =  2.01 # Klay Thompson is 6'7"
steph = 1.91 # Steph Curry is 6'3"
dangelo = 1.95 # D'Angelo Russell is 6'5"
average_adult_height_m = 1.688

# IMPORTANT: For the steps below, DO NOT USE ANY NUMBERS, only function calls␣
 ↪and the names `klay`, `steph`, and `dangelo`.
# STEP 1: Compute the absolute difference between all 3 pairs of heights
# Note: the following 3 lines are helper lines to complete this first step, you␣
 ↪can skip them if you want to combine STEP 1 + STEP 2 in 1 line of code)
klay_steph_diff = klay - steph
klay_dangelo_diff = klay - dangelo
dangelo_steph_diff = dangelo - steph
# STEP 2: Find the smallest of those 3 absolute differences.
min_height_difference = min(abs(klay_steph_diff), abs(klay_dangelo_diff),␣
 ↪abs(dangelo_steph_diff))


min_height_difference # This line is just so you can verify your answer
```

[116]: 0.040000000000000036

[117]:
```
grader.check("q5_1_1")
```

[117]: q5_1_1 results: All test cases passed!

Congratulations, you're done with Lab 1!

Be sure to…

- Run all of the cells and the tests
- Verify that they all pass,
- then choose **Download as PDF via LaTeX** from the **File** menu, correctly name your file, and submit the .pdf file on **canvas**.