software carpentry

Programming with Python

Instructor's Guide

Legend

We are using a dataset with records on inflammation from patients following an arthritis treatment.

We make reference in the lesson that this data is somehow strange. It is strange because it is fabricated! The script used to generate the inflammation data is included as `tools/gen_inflammation.py`.

Overall

This lesson is written as an introduction to Python, but its real purpose is to introduce the single most important idea in programming: how to solve problems by building functions, each of which can fit in a programmer's working memory. In order to teach that, we must teach people a little about the mechanics of manipulating data with lists and file I/O so that their functions can do things they actually care about. Our teaching order tries to show practical uses of every idea as soon as it is introduced; instructors should resist the temptation to explain the "other 90%" of the language as well.

The final example asks them to build a command-line tool that works with the Unix pipe-and-filter model. We do this because it is a useful skill and because it helps learners see that the software they use isn't magical. Tools like `grep` might be more sophisticated than the programs our learners can write at this point in their careers, but it's crucial they realize this is a difference of scale rather than kind.

Explain that we use Python because:

- It's free.
- It has a lot of scientific libraries, and more are constantly being added.
- It has a large scientific user community.
- It's easier for novices to learn than most of the mature alternatives. (Software Carpentry originally used Perl; when we switched, we found that we could cover as much material in two days in Python as we'd covered in three days in Perl, and that retention was higher.)

We do *not* include instructions on running the IPython Notebook in the tutorial because we want to focus on the language rather than the tools. Instructors should, however, walk learners through some basic operations:

- Launch from the command line with `ipython notebook`.
- Create a new notebook.
- Enter code or data in a cell and execute it.
- Explain the difference between `In[#]` and `Out[#]`.

Watching the instructor grow programs step by step is as helpful to learners as anything to do with Python. Resist the urge to update a single cell repeatedly (which is what you'd probably do in real life). Instead, clone the previous cell and write the update in the new copy so that learners have a complete record of how the program grew. Once you've done this, you can say, "Now why don't we just break things into small functions right from the start?"

The discussion of command-line scripts assumes that students understand standard I/O and building filters, which are covered in the lesson on the shell.

Frequently Argued Issues (FAI)

- `import ... as ...` syntax.

  This syntax is commonly used in the scientific Python community; it is explicitly recommended in documentation to `import numpy as np` and `import matplotlib.pyplot as plt`. Despite that, we have decided not to introduce aliasing imports in this novice lesson due to the additional cognitive load it puts on students, despite the typing that it saves. A good summary of arguments for and against can be found in PR #61.

  It is up to you as an individual instructor whether you want to introduce these aliases when you teach this lesson, but we encourage you to please read those arguments thoroughly before deciding one way or the other.

Analyzing Patient Data

Repeating Actions with Loops

Solutions to exercises:

From 1 to N

Using `range`, write a loop that uses `range` to print the first 3 natural numbers.

```
for i in range(1, 4):
   print(i)
```

```
1
2
3
```

Computing powers with loops

Write a loop that calculates the same result as `5 ** 3` using multiplication (and without exponentiation).

```
result = 1
for i in range(0, 3):
   result = result * 5
print(result)
```

```
125
```

Reverse a string

Write a loop that takes a string, and produces a new string with the characters in reverse order.

```
newstring = ''
oldstring = 'Newton'
length_old = len(oldstring)
for char_index in range(length_old):
   newstring = newstring + oldstring[length_old - char_index - 1]
print(newstring)
```

```
'notweN'
```

After discussing these challenges could be a good time to introduce the `b *= 2` syntax.

Storing Multiple Values in Lists

Solutions to exercises:

Turn a string into a list

Use a `for` loop to convert the string `"hello"` into a list of letters:

```
my_list = []
for char in "hello":
    my_list.append(char)
print(my_list)
```

```
["h", "e", "l", "l", "o"]
```

Analyzing Data from Multiple Files

Making Choices

Solutions to exercises:

How many paths?

Which of the following would be printed if you were to run this code? Why did you pick this answer?

```
if 4 > 5:
    print('A')
elif 4 == 5:
    print('B')
elif 4 < 5:
    print('C')
```

C gets printed, because the first two conditions, `4 > 5` and `4 == 5` are not true, but `4 < 5` is true.

What is truth?

After reading and running the code below, explain the rules for which values are considered true and which are considered false.

```
if '':
    print('empty string is true')
if 'word':
    print('word is true')
if []:
    print('empty list is true')
if [1, 2, 3]:
    print('non-empty list is true')
if 0:
    print('zero is true')
if 1:
    print('one is true')
```

First line prints nothing: an empty string is false Second line prints `'word is true'` : a non-empty string is true Third line prints nothing: an empty list is false Fourth line prints `'non-empty list is true'` : a non-empty list is true Fifth line prints nothing: 0 is false Sixth line prints `'one is true'` : 1 is true

Close enough

Write some conditions that print `True` if the variable `a` is within 10% of the variable `b` and `False` otherwise.

```
a = 5
b = 5.1

if abs(a - b) < 0.1 * abs(b):
    print('True')
else:
    print('False')
```

Another possible solution:

```
print(abs(a - b) < 0.1 * abs(b))
```

This works because the boolean objects `True` and `False` have string representations which can be `print`ed.

In-place operators

Write some code that sums the positive and negative numbers in a list separately, using in-place operators.

```
positive_sum = 0
negative_sum = 0
test_list = [3, 4, 6, 1, -1, -5, 0, 7, -8]
for num in test_list:
    if num > 0:
        positive_sum += num
    elif num == 0:
        pass
    else:
        negative_sum += num
print(positive_sum, negative_sum)
```

```
21 -14
```

Here `pass` means “don’t do anything”. In this particular case, it’s not actually needed, since if `num == 0` neither sum needs to change, but it illustrates the use of `elif`.

Tuples and exchanges

Explain what the overall effect of this code is:

```
left = 'L'
right = 'R'

temp = left
left = right
right = temp
```

The code swaps the contents of the variables right and left.

Compare it to:

```
left, right = right, left
```

Do they always do the same thing? Which do you find easier to read?

Yes, although it's possible the internal implementation is different. Answers will vary on which is easier to read.

Creating Functions

Solutions to exercises:

Combining strings

Write a function called `fence` that takes two parameters called `original` and `wrapper` and returns a new string that has the wrapper character at the beginning and end of the original.

```
def fence(original, wrapper):
    return wrapper + original + wrapper
```

Selecting characters from strings

Write a function called `outer` that returns a string made up of just the first and last characters of its input.

```
def outer(input_string):
    return input_string[0] + input_string[-1]
```

Rescaling an array

Write a function `rescale` that takes an array as input and returns a corresponding array of values scaled to lie in the range 0.0 to 1.0. (Hint: If *L* and *H* are the lowest and highest values in the original array, then the replacement for a value *v* should be (*v* – *L*)/(*H* – *L*).)

```
def rescale(input_array):
    L = input_array.min()
    H = input_array.max()
    output_array = (input_array - L) / (H - L)
    return output_array
```

Testing and documenting your function

Run the commands `help(numpy.arange)` and `help(numpy.linspace)` to see how to use these functions to generate regularly-spaced values, then use those values to test your `rescale` function. Once you've successfully tested your function, add a docstring that explains what it does.

```
'''Takes an array as input, and returns a corresponding array scaled so
that 0 corresponds to the minimum and 1 to the maximum value of the input array
.

Examples:
>>> rescale(np.arange(10.0))
array([ 0.        ,  0.11111111,  0.22222222,  0.33333333,  0.44444444,
       0.55555556,  0.66666667,  0.77777778,  0.88888889,  1.        ])
>>> rescale(np.linspace(0, 100, 5))
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
'''
```

Defining defaults

Rewrite the `rescale` function so that it scales data to lie between 0.0 and 1.0 by default, but will allow the caller to specify lower and upper bounds if they want. Compare your implementation to your neighbor's: do the two functions always behave the same way?

```
def rescale(input_array, low_val=0.0, high_val=1.0):
    '''rescales input array values to lie between low_val and high_val'''
    L = input_array.min()
    H = input_array.max()
    intermed_array = (input_array - L) / (H - L)
    output_array = intermed_array * (high_val - low_val) + low_val
    return output_array
```

Variables inside and outside functions

What does the following piece of code display when run - and why?

```
f = 0
k = 0

def f2k(f):
    k = ((f-32)*(5.0/9.0)) + 273.15
    return k

print(f2k(8))
print(f2k(41))
print(f2k(32))

print(k)
```

```
259.81666666666666
287.15
273.15
0
```

`k` is 0 because the `k` inside the function `f2k` doesn't know about the `k` defined outside the function.

Errors and Exceptions

Solutions to exercises:

Reading Error Messages

Read the traceback below, and identify the following pieces of information about it:

1. How many levels does the traceback have?
2. What is the file name where the error occurred?
3. What is the function name where the error occurred?
4. On which line number in this function did the error occurr?

5. What is the type of error?
6. What is the error message?

```
import errors_02
errors_02.print_friday_message()
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-e4c4cbafeeb5> in <module>()
      1 import errors_02
----> 2 errors_02.print_friday_message()

/Users/jhamrick/project/swc/novice/python/errors_02.py in print_friday_message(
)
     13
     14 def print_friday_message():
---> 15     print_message("Friday")

/Users/jhamrick/project/swc/novice/python/errors_02.py in print_message(day)
      9         "sunday": "Aw, the weekend is almost over."
     10     }
---> 11     print(messages[day])
     12
     13

KeyError: 'Friday'
```

---

1. 3 levels
2. `errors_02.py`
3. `print_message`
4. 11
5. `KeyError`
6. There isn't really a message; you're supposed to infer that `Friday` is not a key in `messages`.

Identifying Syntax Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. Is it a `SyntaxError` or an `IndentationError`?
3. Fix the error.
4. Repeat steps 2 and 3, until you have fixed all the errors.

```
def another_function
    print("Syntax errors are annoying.")
     print("But at least python tells us about them!")
    print("So they are usually not too hard to fix.")
```

`SyntaxError` for missing `():` at end of first line, `IndentationError` for mismatch between second and third lines.

Fixed version:

```
def another_function():
    print("Syntax errors are annoying.")
    print("But at least python tells us about them!")
    print("So they are usually not too hard to fix.")
```

Identifying Variable Name Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. What type of `NameError` do you think this is? In other words, is it a string with no quotes, a misspelled variable, or a variable that should have been defined but was not?
3. Fix the error.
4. Repeat steps 2 and 3, until you have fixed all the errors.

```
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (Number % 3) == 0:
        message = message + a
    else:
        message = message + "b"
print(message)
```

---

3 `NameError`s for `number` being misspelled, for `message` not defined, and for `a` not being in quotes.

Fixed version:

```
message = ""
for number in range(10):
    # use a if the number is a multiple of 3, otherwise use b
    if (number % 3) == 0:
        message = message + "a"
    else:
        message = message + "b"
print(message)
```

```
abbabbabba
```

Identifying Item Errors

1. Read the code below, and (without running it) try to identify what the errors are.
2. Run the code, and read the error message. What type of error is it?
3. Fix the error.

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print('My favorite season is ', seasons[4])
```

`IndexError`; the last entry is `seasons[3]`, so `seasons[4]` doesn't make sense.

Fixed version:

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print('My favorite season is ', seasons[-1])
```

Defensive Programming

Solutions to exercises:

Pre- and post-conditions

Suppose you are writing a function called `average` that calculates the average of the numbers in a list. What pre-conditions and post-conditions would you write for it? Compare your answer to your neighbor's: can you think of a function that will pass your tests but not hers or vice versa?

```
# a possible pre-condition:
assert len(input) > 0, 'List length must be non-zero'
# a possible post-condition:
assert input.min() < average < input.max(), 'Average should be between min and
max of input values'
```

Testing assertions

Given a sequence of values, the function `running` returns a list containing the running totals at each index. Explain in words what the assertions in this function check, and for each one, give an example of input that will make that assertion fail.

```
def running(values):
    assert len(values) > 0
    result = [values[0]]
    for v in values[1:]:
        assert result[-1] >= 0
        result.append(result[-1] + v)
        assert result[-1] >= result[0]
    return result
```

- The first assertion checks that the input sequence `values` is not empty. An empty sequence

such as `[]` will make it fail.
- The second assertion checks that the first value in the list is positive. Input such as `[-1,0,2,3]` will make it fail.
- The third assertion checks that the running total always increases. Input such as `[0,1,3,-5,4]` will make it fail.

Fixing and testing

Fix `range_overlap`. Re-run `test_range_overlap` after each change you make.

```
import numpy

def range_overlap(ranges):
    '''Return common overlap among a set of [low, high] ranges.'''
    if len(ranges) == 1: # only one entry, so return it
        return ranges[0]
    lowest = -numpy.inf # lowest possible number
    highest = numpy.inf # highest possible number
    for (low, high) in ranges:
        lowest = max(lowest, low)
        highest = min(highest, high)
    if lowest >= highest: # no overlap
        return None
    else:
        return (lowest, highest)
```

Debugging

Command-Line Programs

Solutions to exercises:

Arithmetic on the command line

Write a command-line program that does addition and subtraction:

```
$ python arith.py add 1 2
```

```
3
```

```
$ python arith.py subtract 3 4
```

```
-1
```

```
# this is code/arith.py
import sys

def main():
    assert len(sys.argv) == 4, 'Need exactly 3 arguments'

    operator = sys.argv[1]
    assert operator in ['add', 'subtract', 'multiply', 'divide'], \
        'Operator is not one of add, subtract, multiply, or divide: bailing out
'
    try:
        operand1, operand2 = float(sys.argv[2]), float(sys.argv[3])
    except ValueError:
        print('cannot convert input to a number: bailing out')
        return

    do_arithmetic(operand1, operator, operand2)

def do_arithmetic(operand1, operator, operand2):

    if operator == 'add':
        value = operand1 + operand2
    elif operator == 'subtract':
        value = operand1 - operand2
    elif operator == 'multiply':
        value = operand1 * operand2
    elif operator == 'divide':
        value = operand1 / operand2
    print(value)

main()
```

Finding particular files

Using the `glob` module introduced earlier, write a simple version of `ls` that shows files in the current directory with a particular suffix. A call to this script should look like this:

```
$ python my_ls.py py
```

```
left.py
right.py
zero.py
```

```
# this is code/my_ls.py
import sys
import glob

def main():
    '''prints names of all files with sys.argv as suffix'''
    assert len(sys.argv) >= 2, 'Argument list cannot be empty'
    suffix = sys.argv[1] # NB: behaviour is not as you'd expect if sys.argv[1]
is *
    glob_input = '*.' + suffix # construct the input
    glob_output = sorted(glob.glob(glob_input)) # call the glob function
    for item in glob_output: # print the output
        print(item)
    return

main()
```

Changing flags

Rewrite `readings.py` so that it uses `-n`, `-m`, and `-x` instead of `--min`, `--mean`, and `--max` respectively. Is the code easier to read? Is the program easier to understand?

```
# this is code/readings-07.py
import sys
import numpy

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['-n', '-m', '-x'], \
           'Action is not one of -n, -m, or -x: ' + action
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)

def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '-n':
        values = data.min(axis=1)
    elif action == '-m':
        values = data.mean(axis=1)
    elif action == '-x':
        values = data.max(axis=1)

    for m in values:
        print(m)

main()
```

Adding a help message

Separately, modify `readings.py` so that if no parameters are given (i.e., no action is specified and no filenames are given), it prints a message explaining how it should be used.

```
# this is code/readings-08.py
import sys
import numpy

def main():
    script = sys.argv[0]
    if len(sys.argv) == 1: # no arguments, so print help message
        print("""Usage: python readings-08.py action filenames
              action must be one of --min --mean --max
              if filenames is blank, input is taken from stdin;
              otherwise, each filename in the list of arguments is processed in
turn""")
        return

    action = sys.argv[1]
    filenames = sys.argv[2:]
    assert action in ['--min', '--mean', '--max'], \
           'Action is not one of --min, --mean, or --max: ' + action
    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)

def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print(m)

main()
```

Adding a default action

Separately, modify `readings.py` so that if no action is given it displays the means of the data.

```
# this is code/readings-09.py
import sys
import numpy

def main():
    script = sys.argv[0]
    action = sys.argv[1]
    if action not in ['--min', '--mean', '--max']: # if no action given
        action = '--mean'    # set a default action, that being mean
        filenames = sys.argv[1:] # start the filenames one place earlier in th
e argv list
    else:
        filenames = sys.argv[2:]

    if len(filenames) == 0:
        process(sys.stdin, action)
    else:
        for f in filenames:
            process(f, action)

def process(filename, action):
    data = numpy.loadtxt(filename, delimiter=',')

    if action == '--min':
        values = data.min(axis=1)
    elif action == '--mean':
        values = data.mean(axis=1)
    elif action == '--max':
        values = data.max(axis=1)

    for m in values:
        print(m)

main()
```

A file-checker

Write a program called `check.py` that takes the names of one or more inflammation data files as arguments and checks that all the files have the same number of rows and columns. What is the best way to test your program?

```
# this is code/check.py
import sys
import numpy

def main():
    script = sys.argv[0]
    filenames = sys.argv[1:]
    if len(filenames) <=1: #nothing to check
        print('Only 1 file specified on input')
    else:
        nrow0, ncol0 = row_col_count(filenames[0])
        print('First file %s: %d rows and %d columns' % (filenames[0], nrow0,
ncol0))
        for f in filenames[1:]:
            nrow, ncol = row_col_count(f)
            if nrow != nrow0 or ncol != ncol0:
                print('File %s does not check: %d rows and %d columns' % (f, n
row, ncol))
            else:
                print('File %s checks' % f)
        return

def row_col_count(filename):
    try:
        nrow, ncol = numpy.loadtxt(filename, delimiter=',').shape
    except ValueError: #get this if file doesn't have same number of rows and c
olumns, or if it has non-numeric content
        nrow, ncol = (0, 0)
    return nrow, ncol

main()
```

Counting lines

Write a program called `line-count.py` that works like the Unix `wc` command:

- If no filenames are given, it reports the number of lines in standard input.
- If one or more filenames are given, it reports the number of lines in each, followed by the total number of lines.

```
# this is code/line-count.py
import sys

def main():
    '''print each input filename and the number of lines in it,
       and print the sum of the number of lines'''
    filenames = sys.argv[1:]
    sum_nlines = 0 #initialize counting variable

    if len(filenames) == 0: # no filenames, just stdin
        sum_nlines = count_file_like(sys.stdin)
        print('stdin: %d' % sum_nlines)
    else:
        for f in filenames:
            n = count_file(f)
            print('%s %d' % (f, n))
            sum_nlines += n
        print('total: %d' % sum_nlines)

def count_file(filename):
    '''count the number of lines in a file'''
    f = open(filename,'r')
    nlines = len(f.readlines())
    f.close()
    return(nlines)

def count_file_like(file_like):
    '''count the number of lines in a file-like object (eg stdin)'''
    n = 0
    for line in file_like:
        n = n+1
    return n

main()
```