functions

July 15, 2016

1 Table of Contents

2 Functions

3 What's a function?

At this point, we've all seen functions, and probably written a few of our own.

Let's start at the beginning though.

```
In [3]: print('Hello World!')
```

```
Hello World!
```

Okay, maybe that was too far back. Let's turn that line into a user-defined function.

```
In [5]: def say_hello():
            print('Hello World!')

        say_hello()

Hello World!
```

Here we've created a function name `say_hello`, and then **called** it on the third line.

But it does the same things as before. What's the point? All we've done here is increase the complexity of our code for no reward.

3.1 What's the function of a function?

Have any of you wondered what the `print()` function does?

- How does it work?
- Where is it defined?

Since python is open source, we can take a little field-trip.

Pretty cool right?! You're now free to stare into the abyss of Python built-in functions.

But do you actually care?

3.2 No.

The python `print()` function would have to be filled with details about Raven to be less relevant to your life.

The power of functions is that the let you **execute code without caring about the details.**

This is particularly worth remembering since most of these examples are going to be really small, and it might seem more reasonable to write the code without the function.

Functions can:

1. Break a program up into logical sub-sections
2. Keep the 'scope' of a variable confined to avoid naming confusion.
3. Allow us to reuse code instead of rewriting it.

We'll touch on each of these throughout the lecture.

4 Parameters

4.1 Parameters are like those significant others . . .

who adopt the personalities of the person they're currently dating.

```
In [12]: def anon(personality, hobby):
             print('Hi! My name is Anon. We just started dating.')
             print('I\'m {} and like {}.'.format(personality, hobby))
             print('')
```

```
In [13]: anon('peppy', 'mixed drinks')
         anon('nerdy', 'video games')
         anon('serious', 'fixing things')

Hi! My name is Anon. We just started dating.
I'm peppy and like mixed drinks.

Hi! My name is Anon. We just started dating.
I'm nerdy and like video games.

Hi! My name is Anon. We just started dating.
I'm serious and like fixing things.
```

Anon's personality and hobby isn't defined by Anon. They're defined by some outside source...

Here's a more practical example.

The code below works fine:

```
In [13]: dna = 'CGATCGCTAGCCTGCACTCAG'
         gc = dna.count('C') + dna.count('G')
         content = 100 * gc / len(dna)
         print('GC content: {:.2f}%'.format(content))

GC content: 61.90%
```

But what if we multiple DNA sequences we wanted to check?

```
In [14]: def gc_content(dna):
             gc = dna.count('C') + dna.count('G')
             content = 100 * gc / len(dna)
             print('GC content: {:.2f}%'.format(content))

In [16]: gc_content('CATGCTCACTGACCCCCCCCCC')
         gc_content('ATTAATATTATAATTAATCCCCCCCC')
         gc_content('ATGCATGCATGCATGCCGTCATGCATGCATCC')

GC content: 72.73%
GC content: 30.77%
GC content: 53.12%
```

Each of these sequences are passed into the function as an **argument**.

4.2 Optional Parameters

If you want, you can give your parameters default values. This can be another way function help create reuseable code.

Here's a simple scenario:
Imagine that you have two lab instruments.
You use the first Monday - Thursday.
It requires a GC content to be input as a percent.
You use the second only on Friday.
It requires GC content as a ratio to five decimal places.

```
In [26]: def gc_content(dna, percent=True):
             gc = dna.count('C') + dna.count('G')
             content = gc / len(dna)
             if percent:
                 content *= 100
                 print('GC content: {:.2f}%'.format(content))
             else:
                 print('GC content: {:.5f}'.format(content))

In [28]: gc_content('ATATCCCGATGCATGCATTAAT')
         gc_content('ATATCCCGATGCATGCATTAAT', False)
```

```
GC content: 36.36%
GC content: 0.36364
```

The first four days of the week, you don't need the optional parameter, since it's set correctly.

5 Quiz

What do the following snippets execute?

```
In [14]: seq = 'AWSDFGTHKLPARTGHLKAI'

         def count_alanine(protein):
             print(protein.count('A'))

         seq = 'PLLIMNHYYTGFRDCSSA'
         count_alanine(seq)
```

```
1
```

- Error
- 3
- 1
- 1 followed by an Error

```
In [30]: def count_amino_acid(protein, aa='A'):
             print(protein.count(aa))

         protein = 'AWSDFGTHKLPARTGHLKAI'
         count_amino_acid(protein)
         count_amino_acid('PLLIMNHYYTGFRDCSSA', 'L')
```

```
3
2
```

- 3 2
- Error
- 2 2
- 3 1

6 Return

Sometimes, you want to get information out of a function.

Finding the length of a sequence is a common, built-in example of this.

```
In [36]: length = len('this could be any sequence')
         print(length)
```

```
26
```

We don't need to go on another excursion, but it's pretty simple to imagine what's going on behind the scenes here.

```
In [37]: def my_len(sequence):
             count = 0
             for i in sequence:
                 count += 1
             return count

         length = my_len('this could be any sequence')
         print(length)
```

```
26
```

The **return** keyword is what's used to send back a variable. Here we are returning `count` and assigning it to `length`.

6.1 A slightly more complicated example

```
In [40]: fasta = """
         >Abhd11os-001|Abhd11os|594|
         TCTGTCTTCTAGCCTCTACCAACTGACAAGTCTCAGTCA
         >Airn-001|Airn|1113|
         GCAAGAAGCACAGCACCGCCAGTTACCACGCAGACATCCTGGGGAACTGAGC
         >Crnde-001|Crnde|1238|
         ATGGAGACCAGAAGCGGTCGGCTGCTGAAGCGGGCGGCTGCGAGCT
         >Csmd2os-001|Csmd2os|615|
         ACAGGGCTCGAGAAAGGGATGGAAAGTGAGTT
         >Dancr-001|Dancr|575|
         TCTCCCGGATGGCTGTATTAACGCAGCGCGCAGCGGCTCGGTCTTTCGGTCCCGGCCTTGGTGG
         """.split()

         def get_fasta_data(fasta):
             headers = []
             seqs = []
             for line in fasta:
                 if line[0] == '>':
                     headers.append(line)
                 else:
                     seqs.append(line)
             return headers, seqs

         headers, seqs = get_fasta_data(fasta)
         print(headers)
```

```
['>Abhd11os-001|Abhd11os|594|', '>Airn-001|Airn|1113|', '>Crnde-001|Crnde|1238|', '
```

7 Quiz

What do the following snippets execute?

```
In [8]: def expected_population_mean(pop_values, expected):
            total = sum(pop_values)
            mean = total/len(pop_values)
            if mean == expected:
                return True
            else:
                return False

        pop = [4, 5, 6, 7, 8]
        result = expected_population_mean(pop, 6)
        print(result)
```

```
True
```

- True
- False
- True and False
- Error

8 Scope

Think of a **scope** as a container for variables.

- The variables do not exist outside of the scope.
- If the scope is destroyed the variables are also destroyed.

Scoping rules are dependent on the language. So the following rules may not be exactly the same in R.

8.1 Local variables cannot be used in the global scope

```
In [54]: def square(x):
             x_squared = x * x

         square(5)
         print(x_squared)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-54-41270b7d312c> in <module>()
  3
  4 square(5)
----> 5 print(x_squared)


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

`x_squared` is only defined in the context of `square()`.

8.2 Local variables cannot be used in other local scopes

```
In [56]: def square(x):
             x_squared = x * x
             return x_squared

         def pythagorean(a, b):
             x_squared = 0
             print(x_squared)
```

```
        a_squared = square(a)
        print(x_squared)
        b_squared = square(b)
        print(x_squared)
        c = (a_squared + b_squared) ** .5
        print(c)

    pythagorean(3,4)
```

```
0
0
0
5.0
```

This may be tricky. Walk through each line to figure out which variable is printing.

8.3 Global variables can be read from a local scope

```
In [3]: COPIES_PER_CELL = 23

        def total_copies(cell_number):
            copies = cell_number*COPIES_PER_CELL
            return copies

        def gene_level(drug_amount):
            gene = (1/2)*drug_amount**2 + COPIES_PER_CELL
            return gene

        print(total_copies(3))
        print(gene_level(123))
```

```
69
7587.5
```

Use these sparingly, and generally as something similar to a physical constant.

8.4 Global variables cannot be modified from a local scope

```
In [5]: DAYS_IN_WEEK = 7

        def change_days_in_week(delta):
            DAYS_IN_WEEK += delta
            return DAYS_IN_WEEK

        print(change_days_in_week(1))
```

```
---------------------------------------------------------------------------

UnboundLocalError                         Traceback (most recent call last)

<ipython-input-5-ef1831d1906a> in <module>()
      5     return DAYS_IN_WEEK
      6
----> 7 print(change_days_in_week(1))


<ipython-input-5-ef1831d1906a> in change_days_in_week(delta)
      2
      3 def change_days_in_week(delta):
----> 4     DAYS_IN_WEEK += delta
      5     return DAYS_IN_WEEK
      6


UnboundLocalError: local variable 'DAYS_IN_WEEK' referenced before assignme
```

That's a lot of rules. Let's go over them.

9 Quiz

What do the following snippets execute?

```
In [11]: # All *interal* function logic is correct.

         def reverse(string):
             return string[::-1]

         def compliment(dna):
             comp_dict = {'A':'T', 'T':'A', 'G':'C', 'C':'G'}
             comp = ''.join(comp_dict[base] for base in dna)
             return comp

         def reverse_compliment(dna):
             dna = reverse(dna)
             dna = compliment(dna)
             return dna

         seq = 'AGCTA'
         print(reverse_compliment(seq))
```

```
TAGCT
```

- ATCGA
- TCGAT
- TAGCT
- Error

TODO: Add more questions.

10 A larger example

We've all written scripts where we start at the beginning and go line by line until we're done.

WARNING: The following code focuses on simplicity and shouldn't be used in practice.

```
In [ ]: def main(h_fasta, m_fasta, h_out, m_out):
            with open(h_fasta) as h_fasta, open(m_fasta) as m_fasta:
                h_fasta = h_fasta.readlines()
                m_fasta = m_fasta.readlines()

            h_names = []
            for line in h_fasta:
                if line[0] == ">":
                    line_ls = line.split("|")
                    name = line_ls[4]
                    h_names.append(name)

            m_names = []
            for line in m_fasta:
                if line[0] == ">":
                    line_ls = line.split("|")
                    name = line_ls[4]
                    m_names.append(name)

            m_names_upper = []
            for name in m_names:
                m_names_upper.append(name.upper())

            h_overlap = []
            m_overlap = []

            for name in m_names_upper:
                if name in h_names:
                    m_pos = m_names_upper.index(name)
                    m_orig_name = m_names[m_pos]
                    m_overlap.append(m_orig_name)
                    h_overlap.append(name)

            with open(h_out, "w") as h_out, open(m_out, "w") as m_out:
```

```
            for name in h_overlap:
                for i, line in enumerate(h_fasta):
                    if name in line:
                        h_out.write(line)
                        h_out.write(h_fasta[i+1])
                        break

            for name in m_overlap:
                for i, line in enumerate(m_fasta):
                    if name in line:
                        m_out.write(line)
                        m_out.write(m_fasta[i+1])
                        break
    main('h_fasta.txt', 'm_fasta.txt', 'h_out.txt', 'm_out.txt')
```

10.1 That function works.

- What does it do?
- Why is there so much repetitiveness?

Add a bit of documentation in the form of well-named functions and **docstrings**.

```
In [ ]: def fasta_name_overlap(h_fasta, m_fasta, h_out, m_out):
            """Writes sequences with corresponding common names to new fasta files.
            with open(h_fasta) as h_fasta, open(m_fasta) as m_fasta:
                h_fasta = h_fasta.readlines()
                m_fasta = m_fasta.readlines()

            h_names = []
            for line in h_fasta:
                if line[0] == ">":
                    line_ls = line.split("|")
                    name = line_ls[4]
                    h_names.append(name)

            m_names = []
            for line in m_fasta:
                if line[0] == ">":
                    line_ls = line.split("|")
                    name = line_ls[4]
                    m_names.append(name)

            m_names_upper = []
            for name in m_names:
                m_names_upper.append(name.upper())

            h_overlap = []
            m_overlap = []
```

```
            for name in m_names_upper:
                if name in h_names:
                    m_pos = m_names_upper.index(name)
                    m_orig_name = m_names[m_pos]
                    m_overlap.append(m_orig_name)
                    h_overlap.append(name)

            with open(h_out, "w") as h_out, open(m_out, "w") as m_out:

                for name in h_overlap:
                    for i, line in enumerate(h_fasta):
                        if name in line:
                            h_out.write(line)
                            h_out.write(h_fasta[i+1])
                            break

                for name in m_overlap:
                    for i, line in enumerate(m_fasta):
                        if name in line:
                            m_out.write(line)
                            m_out.write(m_fasta[i+1])
                            break
        fasta_name_overlap('h_fasta.txt', 'm_fasta.txt', 'h_out.txt', 'm_out.txt')
```

At least now we can have some idea what's going on.
Now we need to get rid of the repeating blocks of code.
We can do that by creating a couple more nicely documented functions.

```
In [ ]: def file_to_list(file_name):
            """Returns a list of lines from a file."""
            with open(file_name) as infile:
                contents = infile.readlines()
            return contents

        def find_names(fasta, upper=False):
            """Finds the common name of gene from a gencode formatted fasta."""
            names = []
            for line in fasta:
                if line[0] == '>':
                    lines_ls = line.split('|')
                    name = line_ls[4]
                    names.append(name)
            if upper:
                upper_names = [n.upper() for n in names]
                return names, upper_names
            else:
                return names
```

```
def fasta_name_overlap(h_fasta, m_fasta, h_out, m_out):
    """Writes sequences with corresponding common names to new fasta files.
    h_fasta = file_to_list(h_fasta)
    m_fasta = file_to_list(m_fasta)

    h_names = find_names(h_fasta)
    m_names, m_upper = find_names(m_fasta, True)

    h_overlap = []
    m_overlap = []

    for name in m_names_upper:
        if name in h_names:
            m_pos = m_names_upper.index(name)
            m_orig_name = m_names[m_pos]
            m_overlap.append(m_orig_name)
            h_overlap.append(name)

    with open(h_out, "w") as h_out, open(m_out, "w") as m_out:

        for name in h_overlap:
            for i, line in enumerate(h_fasta):
                if name in line:
                    h_out.write(line)
                    h_out.write(h_fasta[i+1])
                    break

        for name in m_overlap:
            for i, line in enumerate(m_fasta):
                if name in line:
                    m_out.write(line)
                    m_out.write(m_fasta[i+1])
                    break
fasta_name_overlap('h_fasta.txt', 'm_fasta.txt', 'h_out.txt', 'm_out.txt')
```

11 Our line count is the same...

But that's not the point.

The purpose here is to increase understanding of what this code is doing.

Let's keep going.

```
In [ ]: def file_to_list(file_name):
            """Returns a list of lines from a file."""
            with open(file_name) as infile:
                contents = infile.readlines()
            return contents
```

```
def find_names(fasta, upper=False):
    """Finds the common name of gene from a gencode formatted fasta."""
    names = []
    for line in fasta:
        if line[0] == '>':
            lines_ls = line.split('|')
            name = line_ls[4]
            names.append(name)
    if upper:
        upper_names = [n.upper() for n in names]
        return names, upper_names
    else:
        return names

def get_overlap(h_names, m_names, m_upper):
    """Returns original names found in both fasta files."""
    h_overlap = []
    m_overlap = []
    for name in m_upper:
        if name in h_names:
            m_pos = m_upper.index(name)
            m_orig_name = m_names[m_pos]
            m_overlap.append(m_orig_name)
            h_overlap.append(name)
    return h_overlap, m_overlap

def filter_fasta(out, overlap, fasta):
    """Filters original fasta files based on overlapping names."""
    with open(out, 'w') as outfile:
        for name in overlap:
            for i, line in enumerate(fasta):
                if name in line:
                    out.write(line)
                    out.write(fasta[i+1])
                    break

def fasta_name_overlap(h_fasta, m_fasta, h_out, m_out):
    """Writes sequences with corresponding common names to new fasta files.
    h_fasta = file_to_list(h_fasta)
    m_fasta = file_to_list(m_fasta)

    h_names = find_names(h_fasta)
    m_names, m_upper = find_names(m_fasta, True)

    h_overlap, m_overlap = get_overlap(h_names, m_names, m_upper)

    filter_fasta(h_out, h_overlap, h_fasta)
```

```
        filter_fasta(h_out, h_overlap, h_fasta)

fasta_name_overlap('h_fasta.txt', 'm_fasta.txt', 'h_out.txt', 'm_out.txt')
```

12 What does this code do?

Read the main function.

- Convert files to lists
- Find fasta names
- Get overlapping names
- Filter fasta files

The main function now serves two purposes.

1. Execute all the smaller functions
2. Explains what the code does to users (including your future self).