Data workflows and automation

Overview

Teaching: 40 min
Exercises: 50 min
Questions
- Can I automate operations in Python?
- What are functions and why should I use them?

Objectives
- Describe why for loops are used in Python.
- Employ for loops to automate data analysis.
- Write unique filenames in Python.
- Build reusable code in Python.
- Write functions using conditional statements (if, then, else).

So far, we've used Python and the pandas library to explore and manipulate individual datasets by hand, much like we would do in a spreadsheet. The beauty of using a programming language like Python, though, comes from the ability to automate data processing through the use of loops and functions.

For loops

Loops allow us to repeat a workflow (or series of actions) a given number of times or while some condition is true. We would use a loop to automatically process data that's stored in multiple files (daily values with one file per year, for example). Loops lighten our work load by performing repeated tasks without our direct involvement and make it less likely that we'll introduce errors by making mistakes while processing each file by hand.

Let's write a simple for loop that simulates what a kid might see during a visit to the zoo:

```
>>> animals = ['lion', 'tiger', 'crocodile', 'vulture', 'hippo']
>>> print(animals)
['lion', 'tiger', 'crocodile', 'vulture', 'hippo']

>>> for creature in animals:
...    print(creature)
lion
tiger
crocodile
vulture
hippo
```

The line defining the loop must start with `for` and end with a colon, and the body of the loop must be indented.

In this example, `creature` is the loop variable that takes the value of the next entry in `animals` every time the loop goes around. We can call the loop variable anything we like. After the loop finishes, the loop variable will still exist and will have the value of the last entry in the collection:

```
>>> animals = ['lion', 'tiger', 'crocodile', 'vulture', 'hippo']
>>> for creature in animals:
...     pass

>>> print('The loop variable is now: ' + creature)
The loop variable is now: hippo
```

We are not asking python to print the value of the loop variable anymore, but the for loop still runs and the value of `creature` changes on each pass through the loop. The statement `pass` in the body of the loop just means "do nothing".

Challenge - Loops

1. What happens if we don't include the `pass` statement?
2. Rewrite the loop so that the animals are separated by commas, not new lines (Hint: You can concatenate strings using a plus sign. For example, `print(string1 + string2)` outputs 'string1string2').

Automating data processing using For Loops

The file we've been using so far, `surveys.csv`, contains 25 years of data and is very large. We would like to separate the data for each year into a separate file.

Let's start by making a new directory inside the folder `data` to store all of these files using the module `os`:

```
    import os

    os.mkdir('data/yearly_files')
```

The command `os.mkdir` is equivalent to `mkdir` in the shell. Just so we are sure, we can check that the new directory was created within the `data` folder:

```
>>> os.listdir('data')
['plots.csv',
 'portal_mammals.sqlite',
 'species.csv',
 'survey2001.csv',
 'survey2002.csv',
 'surveys.csv',
 'surveys2002_temp.csv',
 'yearly_files']
```

The command `os.listdir` is equivalent to `ls` in the shell.

In previous lessons, we saw how to use the library pandas to load the species data into memory as a DataFrame, how to select a subset of the data using some criteria, and how to write the DataFrame into a csv file. Let's write a script that performs those three steps in sequence for the year 2002:

```
import pandas as pd

# Load the data into a DataFrame
surveys_df = pd.read_csv('data/surveys.csv')

# Select only data for 2002
surveys2002 = surveys_df[surveys_df.year == 2002]

# Write the new DataFrame to a csv file
surveys2002.to_csv('data/yearly_files/surveys2002.csv')
```

To create yearly data files, we could repeat the last two commands over and over, once for each year of data. Repeating code is neither elegant nor practical, and is very likely to introduce errors into your code. We want to turn what we've just written into a loop that repeats the last two commands for every year in the dataset.

Let's start by writing a loop that simply prints the names of the files we want to create - the dataset we are using covers 1977 through 2002, and we'll create a separate file for each of those years. Listing the filenames is a good way to confirm that the loop is behaving as we expect.

We have seen that we can loop over a list of items, so we need a list of years to loop over. We can get the years in our DataFrame with:

```
>>> surveys_df['year']

0        1977
1        1977
2        1977
3        1977
         ...
35545    2002
35546    2002
35547    2002
35548    2002
```

but we want only unique years, which we can get using the `unique` function which we have already seen.

```
>>> surveys_df['year'].unique()
array([1977, 1978, 1979, 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987,
       1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998,
       1999, 2000, 2001, 2002], dtype=int64)
```

Putting this into our for loop we get

```
>>> for year in surveys_df['year'].unique():
...    filename='data/yearly_files/surveys' + str(year) + '.csv'
...    print(filename)
...
data/yearly_files/surveys1977.csv
data/yearly_files/surveys1978.csv
data/yearly_files/surveys1979.csv
data/yearly_files/surveys1980.csv
data/yearly_files/surveys1981.csv
data/yearly_files/surveys1982.csv
data/yearly_files/surveys1983.csv
data/yearly_files/surveys1984.csv
data/yearly_files/surveys1985.csv
data/yearly_files/surveys1986.csv
data/yearly_files/surveys1987.csv
data/yearly_files/surveys1988.csv
data/yearly_files/surveys1989.csv
data/yearly_files/surveys1990.csv
data/yearly_files/surveys1991.csv
data/yearly_files/surveys1992.csv
data/yearly_files/surveys1993.csv
data/yearly_files/surveys1994.csv
data/yearly_files/surveys1995.csv
data/yearly_files/surveys1996.csv
data/yearly_files/surveys1997.csv
data/yearly_files/surveys1998.csv
data/yearly_files/surveys1999.csv
data/yearly_files/surveys2000.csv
data/yearly_files/surveys2001.csv
data/yearly_files/surveys2002.csv
```

We can now add the rest of the steps we need to create separate text files:

```
# Load the data into a DataFrame
surveys_df = pd.read_csv('data/surveys.csv')

for year in surveys_df['year'].unique():

    # Select data for the year
    surveys_year = surveys_df[surveys_df.year == year]

    # Write the new DataFrame to a csv file
    filename = 'data/yearly_files/surveys' + str(year) + '.csv'
    surveys_year.to_csv(filename)
```

Look inside the `yearly_files` directory and check a couple of the files you just created to confirm that everything worked as expected.

Writing Unique FileNames

Notice that the code above created a unique filename for each year.

```
filename = 'data/yearly_files/surveys' + str(year) + '.csv'
```

Let's break down the parts of this name:

- The first part is simply some text that specifies the directory to store our data file in (data/yearly_files/) and the first part of the file name (surveys): `'data/yearly_files/surveys'`

- We can concatenate this with the value of a variable, in this case `year` by using the plus + sign and the variable we want to add to the file name: `+ str(year)`
- Then we add the file extension as another text string: `+ '.csv'`

Notice that we use single quotes to add text strings. The variable is not surrounded by quotes. This code produces the string `data/yearly_files/surveys2002.csv` which contains the path to the new filename AND the file name itself.

Challenge - Modifying loops

1. Some of the surveys you saved are missing data (they have null values that show up as NaN - Not A Number - in the DataFrames and do not show up in the text files). Modify the for loop so that the entries with null values are not included in the yearly files.
2. Let's say you only want to look at data from a given multiple of years. How would you modify your loop in order to generate a data file for only every 5th year, starting from 1977?
3. Instead of splitting out the data by years, a colleague wants to do analyses each species separately. How would you write a unique csv file for each species?

Building reusable and modular code with functions

Suppose that separating large data files into individual yearly files is a task that we frequently have to perform. We could write a **for loop** like the one above every time we needed to do it but that would be time consuming and error prone. A more elegant solution would be to create a reusable tool that performs this task with minimum input from the user. To do this, we are going to turn the code we've already written into a function.

Functions are reusable, self-contained pieces of code that are called with a single command. They can be designed to accept arguments as input and return values, but they don't need to do either. Variables declared inside functions only exist while the function is running and if a variable within the function (a local variable) has the same name as a variable somewhere else in the code, the local variable hides but doesn't overwrite the other.

Every method used in Python (for example, `print`) is a function, and the libraries we import (say, `pandas`) are a collection of functions. We will only use functions that are housed within the same code that uses them, but it's also easy to write functions that can be used by different programs.

Functions are declared following this general structure:

```
def this_is_the_function_name(input_argument1, input_argument2):

    # The body of the function is indented
    # This function prints the two arguments to screen
    print('The function arguments are:', input_argument1, input_argument2, '(this is do
ne inside the function!)')

    # And returns their product
    return input_argument1 * input_argument2
```

The function declaration starts with the word `def`, followed by the function name and any arguments in parenthesis, and ends in a colon. The body of the function is indented just like loops are. If the function returns something when it is called, it includes a return statement at the end.

This is how we call the function:

```
>>> product_of_inputs = this_is_the_function_name(2,5)
The function arguments are: 2 5 (this is done inside the function!)

>>> print('Their product is:', product_of_inputs, '(this is done outside the function!)
')
Their product is: 10 (this is done outside the function!)
```

Challenge - Functions

1. Change the values of the arguments in the function and check its output
2. Try calling the function by giving it the wrong number of arguments (not 2) or not assigning the function call to a variable (no `product_of_inputs =`)
3. Declare a variable inside the function and test to see where it exists (Hint: can you print it from outside the function?)
4. Explore what happens when a variable both inside and outside the function have the same name. What happens to the global variable when you change the value of the local variable?

We can now turn our code for saving yearly data files into a function. There are many different “chunks” of this code that we can turn into functions, and we can even create functions that call other functions inside them. Let’s first write a function that separates data for just one year and saves that data to a file:

```
def one_year_csv_writer(this_year, all_data):
    """
    Writes a csv file for data from a given year.

    this_year --- year for which data is extracted
    all_data --- DataFrame with multi-year data
    """

    # Select data for the year
    surveys_year = all_data[all_data.year == this_year]

    # Write the new DataFrame to a csv file
    filename = 'data/yearly_files/function_surveys' + str(this_year) + '.csv'
    surveys_year.to_csv(filename)
```

The text between the two sets of triple double quotes is called a docstring and contains the documentation for the function. It does nothing when the function is running and is therefore not necessary, but it is good practice to include docstrings as a reminder of what the code does. Docstrings in functions also become part of their ‘official’ documentation:

```
one_year_csv_writer?
```

```
one_year_csv_writer(2002,surveys_df)
```

We changed the root of the name of the csv file so we can distinguish it from the one we wrote before. Check the `yearly_files` directory for the file. Did it do what you expect?

What we really want to do, though, is create files for multiple years without having to request them one by one. Let’s write another function that replaces the entire For loop by simply looping through a sequence of years and repeatedly calling the function we just wrote, `one_year_csv_writer`:

```
def yearly_data_csv_writer(start_year, end_year, all_data):
    """
    Writes separate csv files for each year of data.

    start_year --- the first year of data we want
    end_year --- the last year of data we want
    all_data --- DataFrame with multi-year data
    """

    # "end_year" is the last year of data we want to pull, so we loop to end_year+1
    for year in range(start_year, end_year+1):
        one_year_csv_writer(year, all_data)
```

Because people will naturally expect that the end year for the files is the last year with data, the for loop inside the function ends at `end_year + 1`. By writing the entire loop into a function, we've made a reusable tool for whenever we need to break a large data file into yearly files. Because we can specify the first and last year for which we want files, we can even use this function to create files for a subset of the years available. This is how we call this function:

```
# Load the data into a DataFrame
surveys_df = pd.read_csv('data/surveys.csv')

# Create csv files
yearly_data_csv_writer(1977, 2002, surveys_df)
```

BEWARE! If you are using IPython Notebooks and you modify a function, you MUST re-run that cell in order for the changed function to be available to the rest of the code. Nothing will visibly happen when you do this, though, because simply defining a function without *calling* it doesn't produce an output. Any cells that use the now-changed functions will also have to be re-run for their output to change.

Challenge- More functions

1. Add two arguments to the functions we wrote that take the path of the directory where the files will be written and the root of the file name. Create a new set of files with a different name in a different directory.
2. How could you use the function `yearly_data_csv_writer` to create a csv file for only one year? (Hint: think about the syntax for `range`)
3. Make the functions return a list of the files they have written. There are many ways you can do this (and you should try them all!): either of the functions can print to screen, either can use a return statement to give back numbers or strings to their function call, or you can use some combination of the two. You could also try using the `os` library to list the contents of directories.
4. Explore what happens when variables are declared inside each of the functions versus in the main (non-indented) body of your code. What is the scope of the variables (where are they visible)? What happens when they have the same name but are given different values?

The functions we wrote demand that we give them a value for every argument. Ideally, we would like these functions to be as flexible and independent as possible. Let's modify the function `yearly_data_csv_writer` so that the `start_year` and `end_year` default to the full range of the data if they are not supplied by the user. Arguments can be given default values with an equal sign in the function declaration. Any arguments in the function without default values (here, `all_data`) is a required argument and MUST come before the argument with default values (which are optional in the function call).

```
def yearly_data_arg_test(all_data, start_year = 1977, end_year = 2002):
    """
    Modified from yearly_data_csv_writer to test default argument values!

    start_year --- the first year of data we want --- default: 1977
    end_year --- the last year of data we want --- default: 2002
    all_data --- DataFrame with multi-year data
    """

    return start_year, end_year


start,end = yearly_data_arg_test (surveys_df, 1988, 1993)
print('Both optional arguments:\t', start, end)

start,end = yearly_data_arg_test (surveys_df)
print('Default values:\t\t\t', start, end)
```

```
Both optional arguments:    1988 1993
Default values:                     1977 2002
```

The “\t” in the `print` statements are tabs, used to make the text align and be easier to read.

But what if our dataset doesn’t start in 1977 and end in 2002? We can modify the function so that it looks for the start and end years in the dataset if those dates are not provided:

```
def yearly_data_arg_test(all_data, start_year = None, end_year = None):
    """
    Modified from yearly_data_csv_writer to test default argument values!

    start_year --- the first year of data we want --- default: None - check all_dat
a
    end_year --- the last year of data we want --- default: None - check all_data
    all_data --- DataFrame with multi-year data
    """

    if not start_year:
        start_year = min(all_data.year)
    if not end_year:
        end_year = max(all_data.year)

    return start_year, end_year


start,end = yearly_data_arg_test (surveys_df, 1988, 1993)
print('Both optional arguments:\t', start, end)

start,end = yearly_data_arg_test (surveys_df)
print('Default values:\t\t\t', start, end)
```

```
Both optional arguments:    1988 1993
Default values:                     1977 2002
```

The default values of the `start_year` and `end_year` arguments in the function `yearly_data_arg_test` are now `None`. This is a build-it constant in Python that indicates the absence of a value - essentially, that the variable exists in the namespace of the function (the directory of variable names) but that it doesn’t correspond to any existing object.

Challenge - Variables

1. What type of object corresponds to a variable declared as `None`? (Hint: create a variable set to `None` and use the function `type()`)
2. Compare the behavior of the function `yearly_data_arg_test` when the arguments have `None` as a default and when they do not have default values.
3. What happens if you only include a value for `start_year` in the function call? Can you write the function call with only a value for `end_year`? (Hint: think about how the function must be assigning values to each of the arguments - this is related to the need to put the arguments without default values before those with default values in the function definition!)

If Loops

The body of the test function now has two conditional loops (if loops) that check the values of `start_year` and `end_year`. If loops execute the body of the loop when some condition is met. They commonly look something like this:

```
a = 5

if a<0: # meets first condition?

    # if a IS less than zero
    print('a is a negative number')

elif a>0: # did not meet first condition. meets second condition?

    # if a ISN'T less than zero and IS more than zero
    print('a is a positive number')

else: # met neither condition

    # if a ISN'T less than zero and ISN'T more than zero
    print('a must be zero!')
```

Which would return:

```
a is a positive number
```

Change the value of `a` to see how this function works. The statement `elif` means "else if", and all of the conditional statements must end in a colon.

The if loops in the function `yearly_data_arg_test` check whether there is an object associated with the variable names `start_year` and `end_year`. If those variables are `None`, the if loops return the boolean `True` and execute whaever is in their body. On the other hand, if the variable names are associated with some value (they got a number in the function call), the if loops return `False` and do not execute. The opposite conditional statements, which would return `True` if the variables were associated with objects (if they had received value in the function call), would be `if start_year` and `if end_year`.

As we've written it so far, the function `yearly_data_arg_test` associates values in the function call with arguments in the function definition just based in their order. If the function gets only two values in the function call, the first one will be associated with `all_data` and the second with `start_year`, regardless of what we intended them to be. We can get around this problem by calling the function using keyword arguments, where each of the arguments in the function definition is associated with a keyword and the function call passes values to the function using these keywords:

```
def yearly_data_arg_test(all_data, start_year = None, end_year = None):
    """
    Modified from yearly_data_csv_writer to test default argument values!

    start_year --- the first year of data we want --- default: None - check all_dat
a
    end_year --- the last year of data we want --- default: None - check all_data
    all_data --- DataFrame with multi-year data
    """

    if not start_year:
        start_year = min(all_data.year)
    if not end_year:
        end_year = max(all_data.year)

    return start_year, end_year


start,end = yearly_data_arg_test (surveys_df)
print('Default values:\t\t\t', start, end)

start,end = yearly_data_arg_test (surveys_df, 1988, 1993)
print('No keywords:\t\t\t', start, end)

start,end = yearly_data_arg_test (surveys_df, start_year = 1988, end_year = 1993)
print('Both keywords, in order:\t', start, end)

start,end = yearly_data_arg_test (surveys_df, end_year = 1993, start_year = 1988)
print('Both keywords, flipped:\t\t', start, end)

start,end = yearly_data_arg_test (surveys_df, start_year = 1988)
print('One keyword, default end:\t', start, end)

start,end = yearly_data_arg_test (surveys_df, end_year = 1993)
print('One keyword, default start:\t', start, end)
```

```
Default values:                     1977 2002
No keywords:                        1988 1993
Both keywords, in order:    1988 1993
Both keywords, flipped:             1988 1993
One keyword, default end:   1988 2002
One keyword, default start: 1977 1993
```

Challenge - Modifying functions

1. Rewrite the `one_year_csv_writer` and `yearly_data_csv_writer` functions to have keyword arguments with default values
2. Modify the functions so that they don't create yearly files if there is no data for a given year and display an alert to the user (Hint: use conditional statements and if loops to do this. For an extra challenge, use `try` statements!)
3. The code below checks to see whether a directory exists and creates one if it doesn't. Add some code to your function that writes out the CSV files, to check for a directory to write to.

```
        if 'dir_name_here' in os.listdir('.'):
            print('Processed directory exists')
        else:
            os.mkdir('dir_name_here')
            print('Processed directory created')
```

1. The code that you have written so far to loop through the years is good, however it is not necessarily reproducible with different datasets. For instance, what happens to the code if we have additional years of data in our CSV files? Using the tools that you learned in the previous activities, make a list of all years represented in the data. Then create a loop to process your data, that begins at the earliest year and ends at the latest year using that list.

HINT: you can create a loop with a list as follows: `for years in year_list:`

Key Points

(../04-merging-data/)

(../06-visualization-ggplot-python/)

Copyright © 2016 Data Carpentry (http://datacarpentry.org)

Source (https://github.com/datacarpentry/python-ecology-lesson/) / Contributing (https://github.com/datacarpentry/python-ecology-lesson/blob/gh-pages/CONTRIBUTING.md) / Cite (https://github.com/datacarpentry/python-ecology-lesson/blob/gh-pages/CITATION) / Contact ()