Cheatsheets / **Python for Data Science: Working with Data**

Custom Data Methods

Python Functions

Python functions are custom blocks of code that transform **inputs** into **outputs**. For example, the `round()` function transforms an input number into a rounded version:

```
round(3.14)
# Output: 3
```

```
# Python function syntax
def function_name(input_parameters):
    <indented code to copute output>
    return function_output
```

Python Function Indentation

The code that a function executes must be indented after the function `def` line. The standard indentation is to consistently use four spaces or one tab for each line of code. But the key requirement is that all lines for the function are indented the same amount.

The example function `squared_difference` performs operations to compute the squared difference between two numbers. Each line of code needed for the calculation is indented consistently by four spaces.

```
# Function that computes the squared
difference of two numbers
def squared_difference(numbers):
    # code blocks indented by four
spaces
    diff = numbers[0] - numbers[1]
    squared_diff = diff**2
    return squared_diff

squared_difference([3,1])
# Output: 4
```

Python Function Output

The `return` statement in a Python function determines the output of the function. The output can be a value on its own or a variable storing a value.
Multiple output values can be returned by specifying each output separated by a comma:

```
def function(input):
    <indented code>
    return output1, output2
```

```
def find_min_max(numbers):
    min_value = np.min(numbers)
    max_value = np.max(numbers)
    return min_value, max_value

minimum, maximum =
find_min_max([3,6,2,5,1])
print(minimum)
# Output: 1
print(maximum)
# Output: 6
```

Python Functions with Multiple Inputs

Python functions can have multiple inputs, using a comma to separate each input inside the function parentheses.

```
def function(input1, input2):
```

```
# Function that computes a multivariate
equation
def line(x, m, b):
    y = m*x + b
    return y
```

Python Functions with Default Arguments

Function inputs can have **default values**, to be used if the user does not provide input. Default values are assigned during definition by placing an `=` sign after the input parameter name followed by the default value.
The example function `line` takes in three input parameters `x`, `m`, and `b`. When calling `line` without specifying a value for `b`, the function defaults to using `b=0`.

```
# Function that computes a multivariate
equation
# Default value b=0
def line(x, m, b=0):
    y = m*x + b
    return y

line(x=2,m=1)
# Output: 2
```

Calling a Python Function

To use/call a Python function, write the function name followed by parentheses:

```
name()
```

If the function has inputs, specify the inputs in the same order as in the function definition, or by using the input parameter name/keyword (see code snippet for examples).

```
# Function that computes a mathematical
formula
def equation(a, b, c=0):
    y = 4*a + 2*b + c
    return y

# Calling w/ ordered arguments
equation(1,2,2)
# Output: 10

# Calling w/ parameter keywords
equation(b=2.c=2,a=1)
# Output: 10
```

Pandas `.apply()` Method on GroupBy Objects

The `.apply()` method can apply custom aggregation functions to a GroupBy.
In the code snippet, we've written a function `count_no_goals` that takes a column as input and counts the number of entries with the value `0`.
We have then applied that to the `results` DataFrame grouped by the `tournament` column.
This gives us a count of the number of games in each tournament with no goals.
results

year	home_team	away_team	total
2009	Czech Republic	Northern Ireland	0
2012	Egypt	Mauritania	3
2015	Turkey	Latvia	2

```
def count_no_goals(column):
    return (column == 0).sum()

matches_zero =
results.groupby('tournament')\

['total_goals'].apply(count_no_goals)
```

Pandas `.apply()` Method Across Rows or Columns

The `.apply()` method can apply functions across either the rows or columns of a DataFrame using the `axis` keyword where

- `axis=1` applies the function across the **rows**
- `axis=0` applies the function to each **column**

Here we apply the `sum` function to `df` where `row_sum` is the output of the function applied across the rows and `column_sum` is the output of the function applied to each column:

	A	B	row_sum
0	1	3	**4**
1	2	4	**6**
column_sum	**3**	**7**	

```
df = pd.DataFrame({'A':[1,2], 'B':
[3,4]})

# Sum the values across each row
df.apply(sum, axis=1)

# Sum the values in each column
df.apply(sum, axis=0)
```

Custom Functions vs Built-In Methods

While applying custom functions using the `.apply()` method is very flexible, it is oftentimes *much slower* than using built-in pandas methods, especially with bigger datasets. When building data pipelines to clean, pre-process, and model data, it is important to evaluate the advantages and disadvantages of using custom functions or built-in methods for your data task.

Print Share ▾