Understanding and Writing Functions in Python

Margaret Gratian

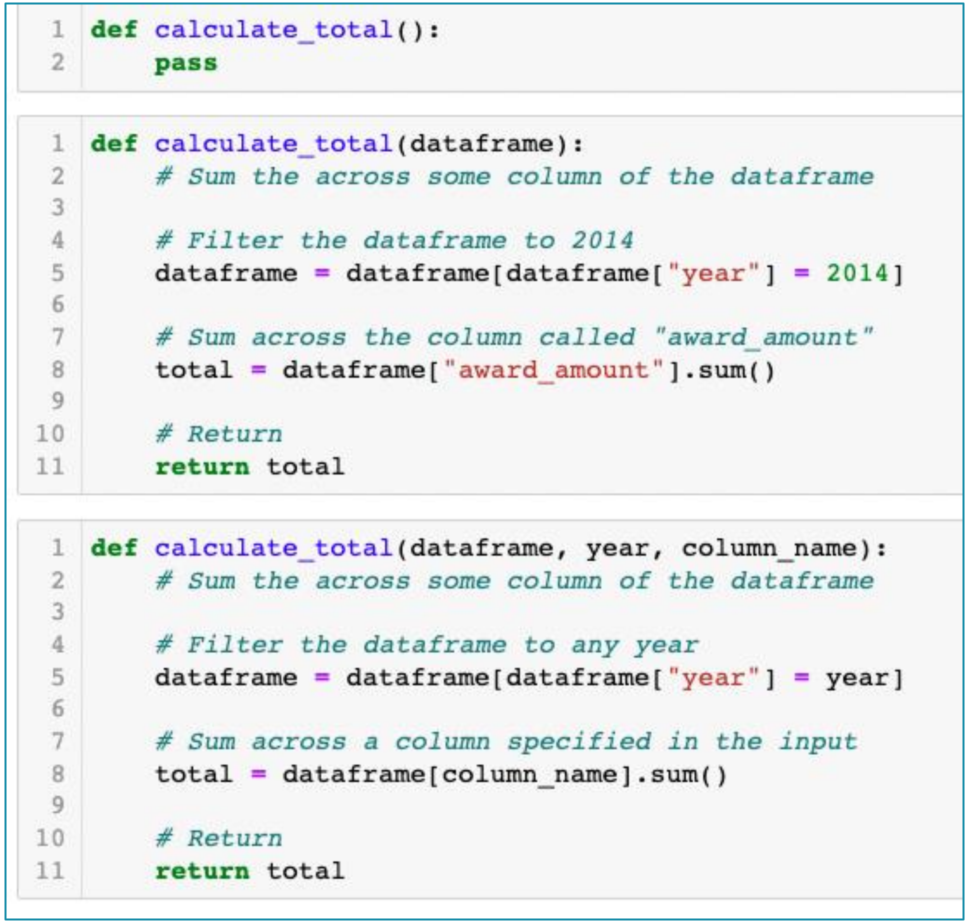
NATIONAL CANCER INSTITUTE

12/22/2022

Defining Functions

- A function is a block of code that performs a task, generally taking some input and returning some output
  - Python comes with many built-in functions for standard tasks, e.g., len(), print()
- In other programming languages, functions may be called methods, subroutines, procedures, etc.
- Functions reduce duplication of code and make code more reusable
- Functions enable implementation of two fundamental concepts in software development:
  - Modularization – break code into smaller, logical components
  - Abstraction – generalizing a problem into a more broadly applicable solution; abstracting away the specific details

Thinking about Modularity and Abstraction



1

```python
def calculate_total():
    pass
```

2

```python
def calculate_total(dataframe):
    # Sum the across some column of the dataframe

    # Filter the dataframe to 2014
    dataframe = dataframe[dataframe["year"] = 2014]

    # Sum across the column called "award_amount"
    total = dataframe["award_amount"].sum()

    # Return
    return total
```

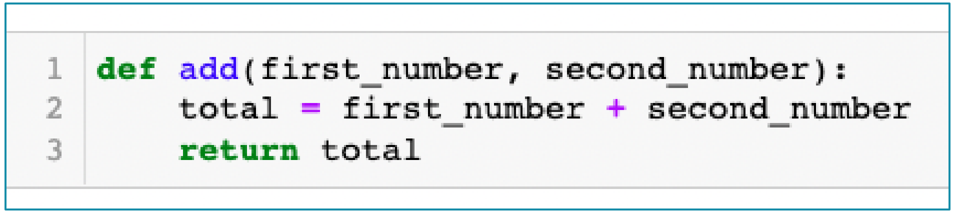
3

```python
def calculate_total(dataframe, year, column_name):
    # Sum the across some column of the dataframe

    # Filter the dataframe to any year
    dataframe = dataframe[dataframe["year"] = year]

    # Sum across a column specified in the input
    total = dataframe[column_name].sum()

    # Return
    return total
```

NATIONAL CANCER INSTITUTE
Center for Research Strategy

Components of a Python Function

- Keyword "**def**" to introduce a function definition
- Function **name** (how you are going to refer to the function)
  - The name "add"
- List of formal **parameters** (the input)
  - first_number and second_number in the example
- Function body (the work)
  - The sum calculation
- A **return** statement (the output)
  - The result of the sum calculation
  - An explicit return statement is not required



```python
def add(first_number, second_number):
    total = first_number + second_number
    return total
```

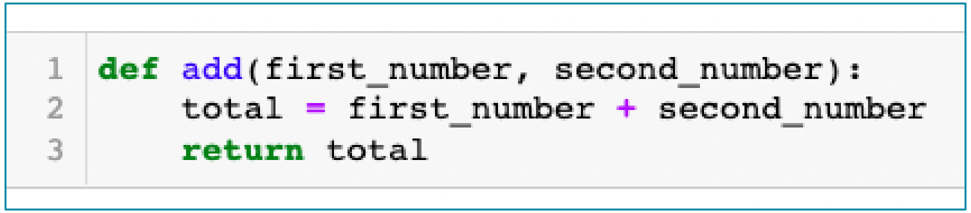
Defining vs. Executing a Function

- Defining a function is not the same as using the function
- When you use a function, you are *calling* or *invoking* it
- You pass *arguments* to the function, which are effectively assigned to the function *parameters* (like variables) during *execution* of the function

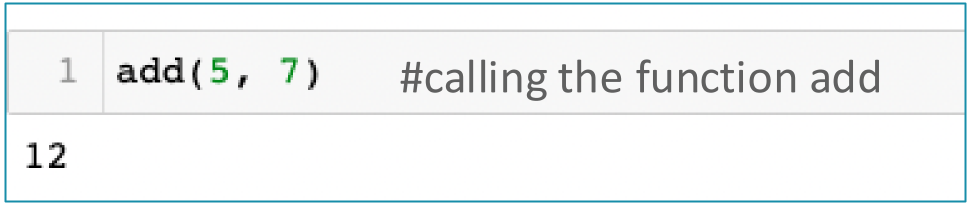
parameters arguments

first_number = 5

second_number = 7



```python
def add(first_number, second_number):
    total = first_number + second_number
    return total
```



```python
add(5, 7)      #calling the function add
```

```
12
```

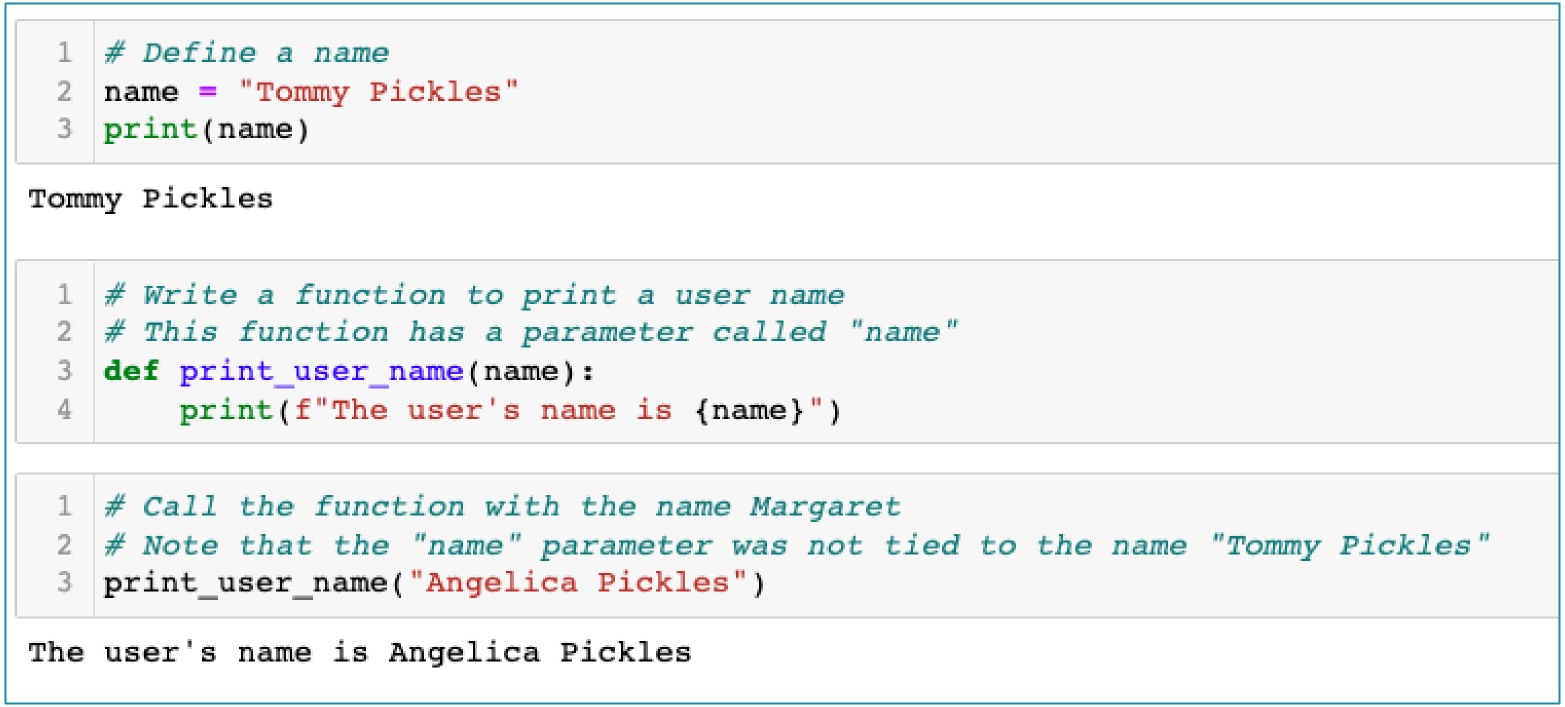
Scope

- The software development concept of *scope* refers to where a name (e.g., the name of a variable or function) is valid and *bound* to a particular entity
  - “x = 2” has bound the variable name x to the value 2
- Python uses the concept of namespaces to maintain a record of how names correspond to values
- If you’ve written a bit of Python, you’ve likely already encountered namespaces in practice
  - For example, when you call “print(“hello)” Python is using namespaces to determine what code print() refers to!
  - When you define a variable in a Jupyter notebook, you are adding that to the current namespace!

Scope in Python Functions

- When a Python function executes, a new namespace is created
- When the function executes, Python will first check the *local* function namespace before looking at other namespaces
- This allows you to use names you may have previously used, without conflict
- This can also be a source of errors if you don't realize what is or is not in scope!
  - If you see a "NameError exception" as an error, you likely have used a variable name that Python cannot find!

Scope in Python Functions



```
# Define a name
name = "Tommy Pickles"
print(name)
```

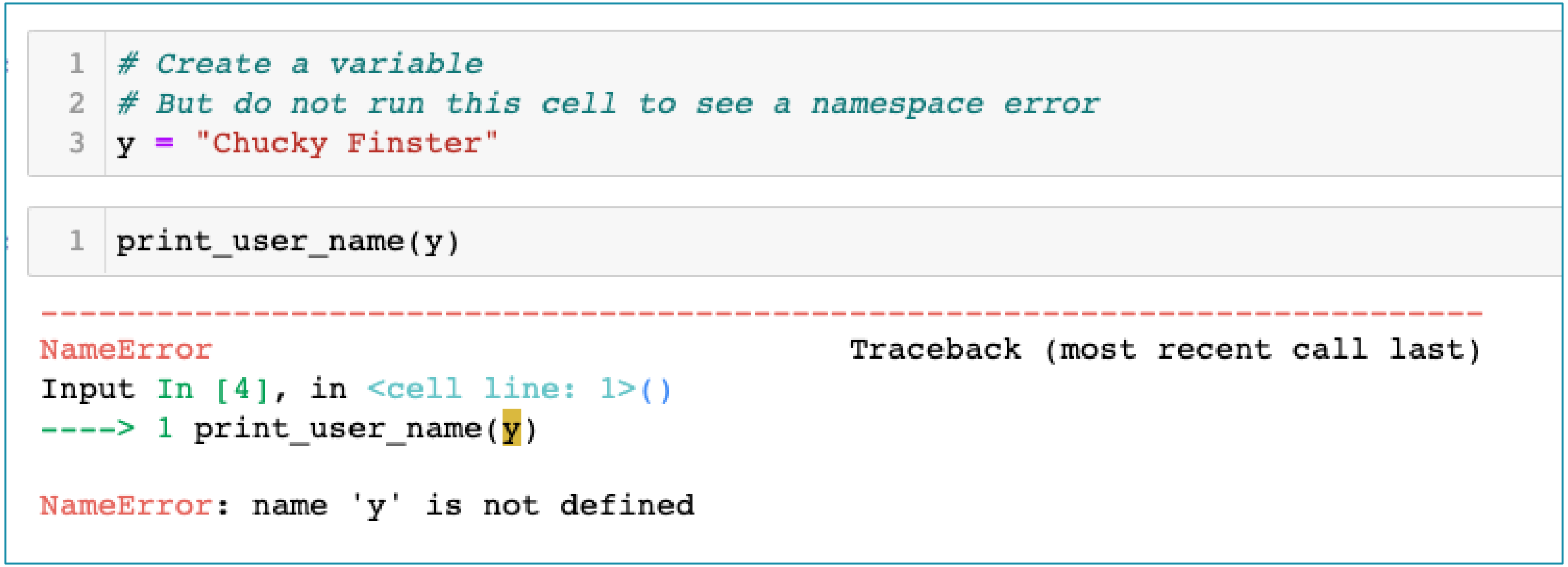
```
Tommy Pickles
```

```
# Write a function to print a user name
# This function has a parameter called "name"
def print_user_name(name):
    print(f"The user's name is {name}")
```

```
# Call the function with the name Margaret
# Note that the "name" parameter was not tied to the name "Tommy Pickles"
print_user_name("Angelica Pickles")
```

```
The user's name is Angelica Pickles
```

Namespace and Scope Errors



```python
# Create a variable
# But do not run this cell to see a namespace error
y = "Chucky Finster"
```

```python
print_user_name(y)
```

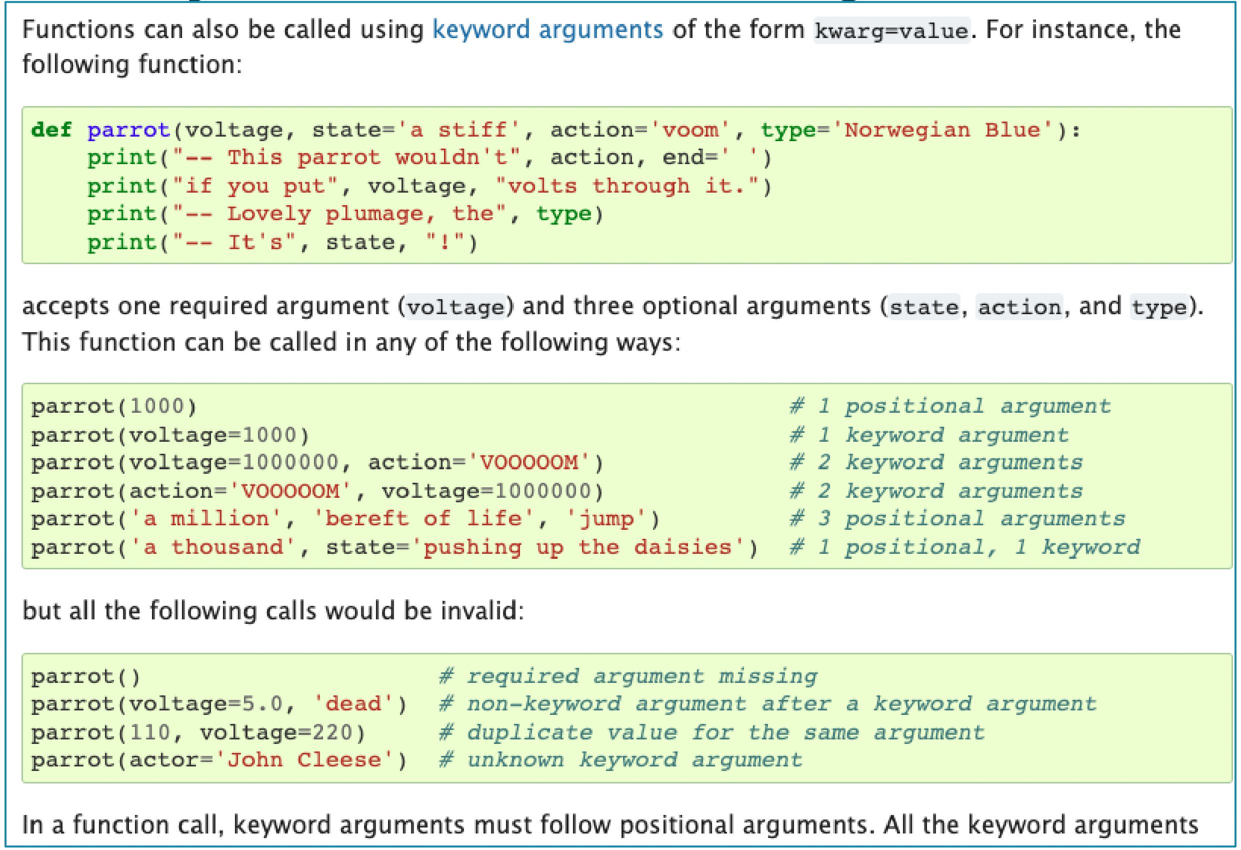
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 print_user_name(y)

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

Behind the scenes, Python was checking the available namespaces to find y! The cell was never run, and there was no other variable called y in the notebook, nor is there a built-in Python variable called y.

Types of Function Parameters

- There are different ways function parameters can be specified, such as:
  - Positional parameters
  - Keyword parameters
- Keyword parameters may have default values that are used if a user does not specify a different one

Positional, Keyword, and **Default** Arguments



Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

but all the following calls would be invalid:

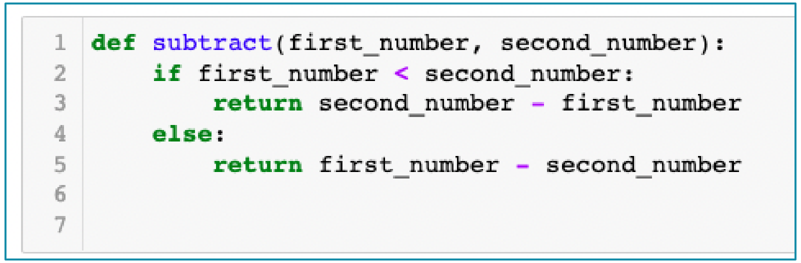
```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments

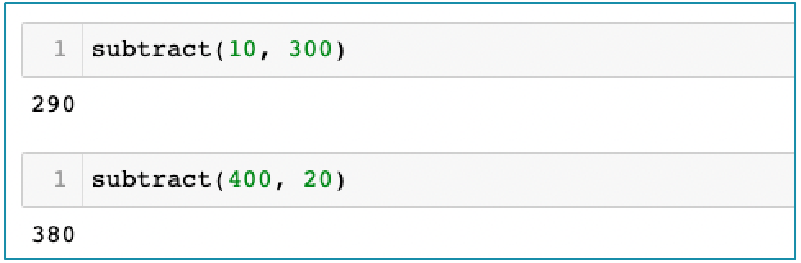
Source: https://docs.python.org/3/tutorial/controlflow.html#default-argument-values

Return Statements

- The “return” statement in Python terminates the function and allows data produced by the function to be returned back to the *caller*
- You can return whatever you want – a string, a DataFrame, a dictionary
- You can even return multiple things…but if you find yourself returning many different things, it may be time to learn more about data structures and some advanced concepts in object oriented programming :)
- If you don’t explicitly name a return function, Python will return None for you behind the scenes



```python
def subtract(first_number, second_number):
    if first_number < second_number:
        return second_number - first_number
    else:
        return first_number - second_number


```



```
subtract(10, 300)
```

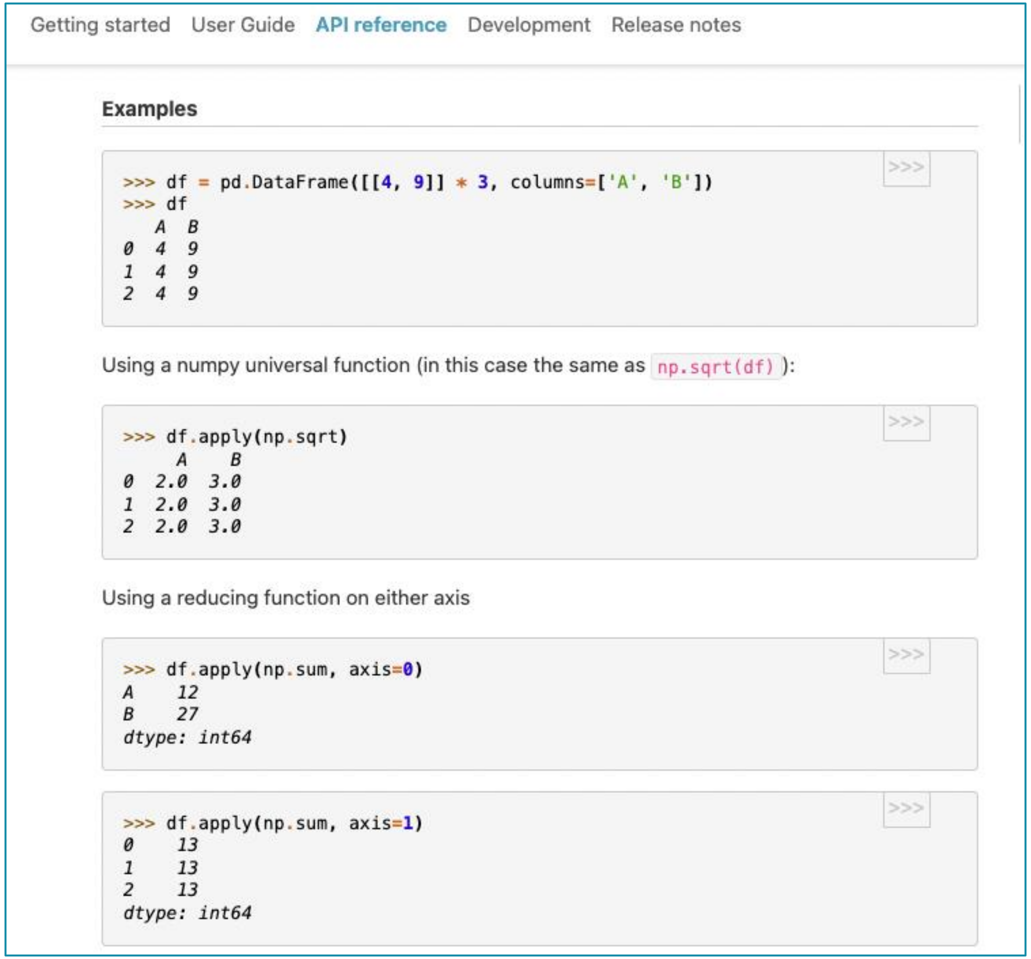
```
290
```

```
subtract(400, 20)
```

```
380
```

Using Functions with DataFrames

- What if you want to use a function to modify a DataFrame?
- A common way to use a function on a DataFrame is to use the .apply() method, which lets you apply a function across an axis (row or column) of a DataFrame
- If axis=1, the function is applied to each row
- If axis=0, the function is applied to each column



Examples

```
>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

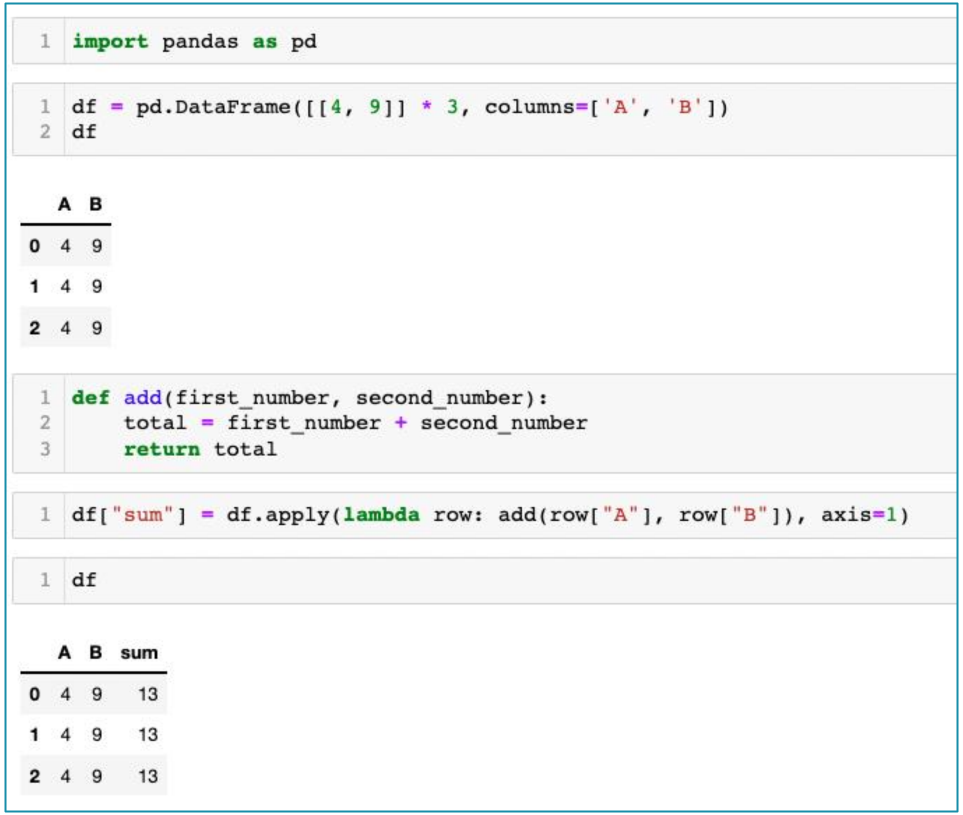
Source: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html

Lambda Functions

- What if you have a custom function you've written and need to use specific rows or columns of your DataFrame as input?
- To solve this problem when working with the Pandas library and DataFrames, you may have encountered the lambda function
- Lambda functions have their origins in the concept of lambda calculus[1] and other programming languages
- In Python, their main difference from other functions is that they are anonymous – they do not need a name!
- In Pandas, they provide a valuable way to apply a custom function to a DataFrame

[1] For more: https://en.wikipedia.org/wiki/Lambda_calculus

Using Lambda Functions with DataFrames



```python
import pandas as pd
```

```python
df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
df
```

```
   A  B
0  4  9
1  4  9
2  4  9
```

```python
def add(first_number, second_number):
    total = first_number + second_number
    return total
```

```python
df["sum"] = df.apply(lambda row: add(row["A"], row["B"]), axis=1)
```

```python
df
```

```
   A  B  sum
0  4  9   13
1  4  9   13
2  4  9   13
```

Note that this example is just to demo the concept of lambda functions.
Columns can also be summed in Pandas as df["sum"] = df[“A”] + df[“B”].

NATIONAL CANCER INSTITUTE
Center for Research Strategy

www.cancer.gov www.cancer.gov/espanol