

User-defined functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

You'll learn:

- Define functions without parameters
- Define functions with one parameter
- Define functions that return a value
- Later: multiple arguments, multiple return values

Built-in functions

- `str()`

```
x = str(5)
```

```
print(x)
```

```
'5'
```

```
print(type(x))
```

```
<class 'str'>
```

Defining a function

```
def square():      # <- Function header
    new_value = 4 ** 2    # <- Function body
    print(new_value)
square()
```

16

Function parameters

```
def square(value):  
    new_value = value ** 2  
    print(new_value)
```

```
square(4)
```

16

```
square(5)
```

25

Return values from functions

- Return a value from a function using return

```
def square(value):  
    new_value = value ** 2  
    return new_value  
  
num = square(4)  
  
print(num)
```

16

Docstrings

- Docstrings describe what your function does
- Serve as documentation for your function
- Placed in the immediate line after the function header
- In between triple double quotes `"""`

```
def square(value):  
    """Return the square of a value."""  
    new_value = value ** 2  
    return new_value
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Multiple Parameters and Return Values

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Multiple function parameters

- Accept more than 1 parameter:

```
def raise_to_power(value1, value2):  
    """Raise value1 to the power of value2."""  
    new_value = value1 ** value2  
    return new_value
```

- Call function: # of arguments = # of parameters

```
result = raise_to_power(2, 3)  
  
print(result)
```

8

A quick jump into tuples

- Make functions return multiple values: Tuples!
- Tuples:
 - Like a list - can contain multiple values
 - Immutable - can't modify values!
 - Constructed using parentheses ()

```
even_nums = (2, 4, 6)  
  
print(type(even_nums))
```

```
<class 'tuple'>
```

Unpacking tuples

- Unpack a tuple into several variables:

```
even_nums = (2, 4, 6)
```

```
a, b, c = even_nums
```

```
print(a)
```

```
2
```

```
print(b)
```

```
4
```

```
print(c)
```

```
6
```

Accessing tuple elements

- Access tuple elements like you do with lists:

```
even_nums = (2, 4, 6)
```

```
print(even_nums[1])
```

```
4
```

```
second_num = even_nums[1]
```

```
print(second_num)
```

```
4
```

- Uses zero-indexing

Returning multiple values

```
def raise_both(value1, value2):  
    """Raise value1 to the power of value2  
    and vice versa."""  
  
    new_value1 = value1 ** value2  
    new_value2 = value2 ** value1  
  
    new_tuple = (new_value1, new_value2)  
  
    return new_tuple
```

```
result = raise_both(2, 3)  
  
print(result)
```

```
(8, 9)
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Bringing it all together

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

You've learned:

- How to write functions
 - Accept multiple parameters
 - Return multiple values
- Up next: Functions for analyzing Twitter data

Basic ingredients of a function

- Function Header

```
def raise_both(value1, value2):
```

- Function body

```
    """Raise value1 to the power of value2  
    and vice versa."""
```

```
    new_value1 = value1 ** value2  
    new_value2 = value2 ** value1
```

```
    new_tuple = (new_value1, new_value2)
```

```
    return new_tuple
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Congratulations!

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Next chapters:

- Functions with default arguments
- Functions that accept an arbitrary number of parameters
- Nested functions
- Error-handling within functions
- More function use in data science!

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Scope and user-defined functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Crash course on scope in functions

- Not all objects are accessible everywhere in a script
- Scope - part of the program where an object or name may be accessible
 - Global scope - defined in the main body of a script
 - Local scope - defined inside a function
 - Built-in scope - names in the pre-defined built-ins module

Global vs. local scope (1)

```
def square(value):  
    """Returns the square of a number."""  
    new_val = value ** 2  
    return new_val  
  
square(3)
```

9

new_val

```
<hr />-----  
NameError                                Traceback (most recent call last)  
<ipython-input-3-3cc6c6de5c5c> in <module>()  
<hr />-> 1 new_value  
NameError: name 'new_val' is not defined
```

Global vs. local scope (2)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    new_val = value ** 2
    return new_val

square(3)
```

9

new_val

10

Global vs. local scope (3)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    new_value2 = new_val ** 2
    return new_value2

square(3)
```

100

```
new_val = 20

square(3)
```

400

Global vs. local scope (4)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    global new_val
    new_val = new_val ** 2
    return new_val

square(3)
```

100

new_val

100

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Nested functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Nested functions (1)

```
def outer( ... ):
    """ ... """
    x = ...

    def inner( ... ):
        """ ... """
        y = x ** 2
    return ...
```

Nested functions (2)

```
def mod2plus5(x1, x2, x3):  
    """Returns the remainder plus 5 of three values."""  
  
    new_x1 = x1 % 2 + 5  
    new_x2 = x2 % 2 + 5  
    new_x3 = x3 % 2 + 5  
  
    return (new_x1, new_x2, new_x3)
```


Nested functions (3)

```
def mod2plus5(x1, x2, x3):  
    """Returns the remainder plus 5 of three values."""  
  
    def inner(x):  
        """Returns the remainder plus 5 of a value."""  
        return x % 2 + 5  
  
    return (inner(x1), inner(x2), inner(x3))
```

```
print(mod2plus5(1, 2, 3))
```

```
(6, 5, 6)
```

Returning functions

```
def raise_val(n):  
    """Return the inner function."""  
  
    def inner(x):  
        """Raise x to the power of n."""  
        raised = x ** n  
        return raised  
  
    return inner
```

```
square = raise_val(2)  
cube = raise_val(3)  
print(square(2), cube(4))
```

4 64

Using nonlocal

```
def outer():  
    """Prints the value of n."""  
    n = 1  
  
    def inner():  
        nonlocal n  
        n = 2  
        print(n)  
  
    inner()  
    print(n)
```

```
outer()
```

```
2
```

```
2
```

Scopes searched

- Local scope
- Enclosing functions
- Global
- Built-in

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Default and flexible arguments

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

You'll learn:

- Writing functions with default arguments
- Using flexible arguments
 - Pass any number of arguments to a functions

Add a default argument

```
def power(number, pow=1):  
    """Raise number to the power of pow."""  
    new_value = number ** pow  
    return new_value
```

```
power(9, 2)
```

```
81
```

```
power(9, 1)
```

```
9
```

```
power(9)
```

```
9
```


Flexible arguments: *args (1)

```
def add_all(*args):  
    """Sum all values in *args together."""  
  
    # Initialize sum  
    sum_all = 0  
  
    # Accumulate the sum  
    for num in args:  
        sum_all += num  
  
    return sum_all
```

Flexible arguments: *args (2)

```
add_all(1)
```

```
1
```

```
add_all(1, 2)
```

```
3
```

```
add_all(5, 10, 15, 20)
```

```
50
```

Flexible arguments: ****kwargs**

```
print_all(name="Hugo Bowne-Anderson", employer="DataCamp")
```

```
name: Hugo Bowne-Anderson  
employer: DataCamp
```

Flexible arguments: **kwargs

```
def print_all(**kwargs):  
    """Print out key-value pairs in **kwargs."""  
  
    # Print out the key-value pairs  
    for key, value in kwargs.items():  
        print(key + ": " + value)
```

```
print_all(name="dumbledore", job="headmaster")
```

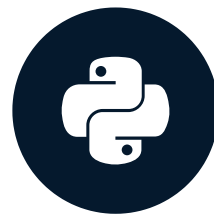
```
job: headmaster  
name: dumbledore
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Bringing it all together

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Next exercises:

- Generalized functions:
 - Count occurrences for any column
 - Count occurrences for an arbitrary number of columns

Add a default argument

```
def power(number, pow=1):  
    """Raise number to the power of pow."""  
    new_value = number ** pow  
    return new_value
```

```
power(9, 2)
```

```
81
```

```
power(9)
```

```
9
```


Flexible arguments: *args (1)

```
def add_all(*args):  
    """Sum all values in *args together."""  
  
    # Initialize sum  
    sum_all = 0  
  
    # Accumulate the sum  
    for num in args:  
        sum_all = sum_all + num  
  
    return sum_all
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Lambda functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Lambda functions

```
raise_to_power = lambda x, y: x ** y
```

```
raise_to_power(2, 3)
```

```
8
```

Anonymous functions

- Function map takes two arguments: `map(func, seq)`
- `map()` applies the function to ALL elements in the sequence

```
nums = [48, 6, 9, 21, 1]

square_all = map(lambda num: num ** 2, nums)

print(square_all)
```

```
<map object at 0x103e065c0>
```

```
print(list(square_all))
```

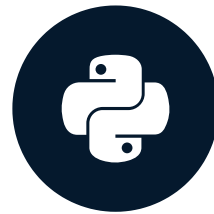
```
[2304, 36, 81, 441, 1]
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

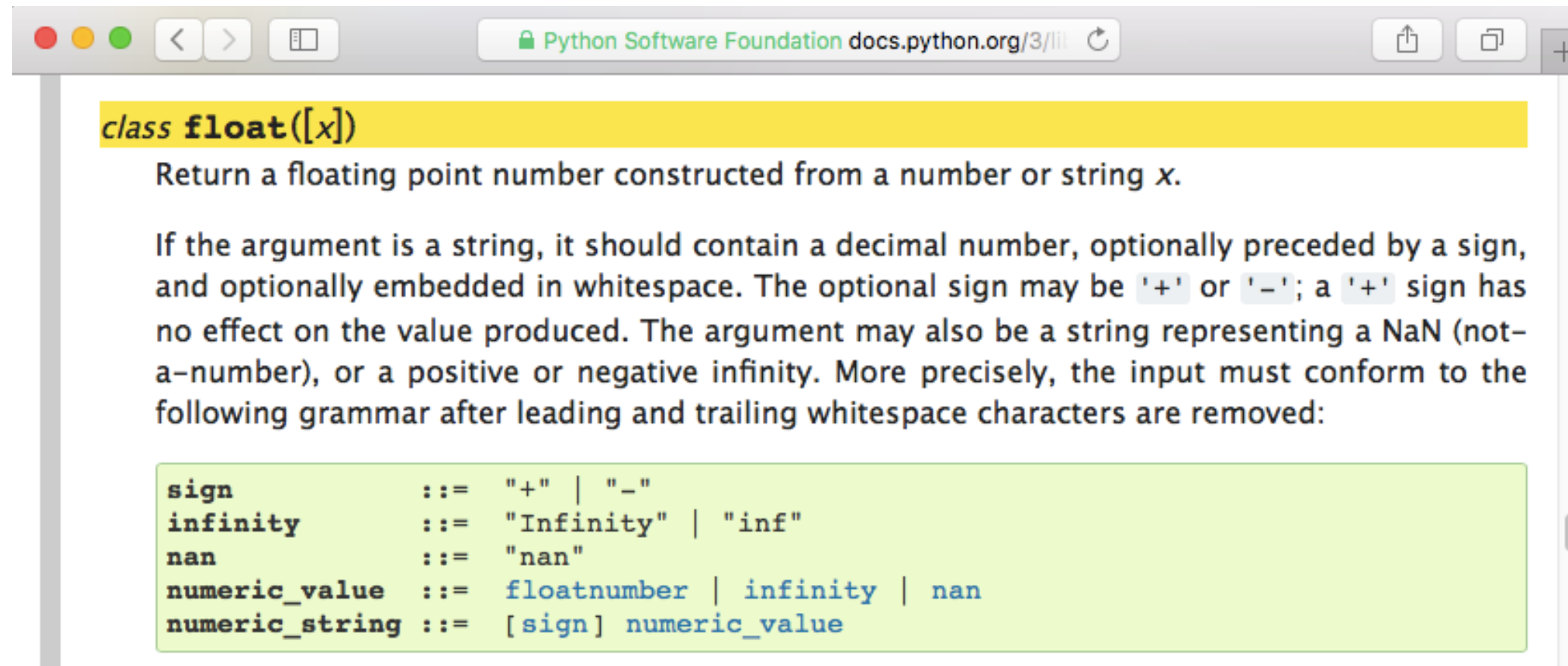
Introduction to error handling

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

The float() function



The screenshot shows a web browser window with the URL `docs.python.org/3/`. The page content is as follows:

`class float([x])`

Return a floating point number constructed from a number or string `x`.

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or a positive or negative infinity. More precisely, the input must conform to the following grammar after leading and trailing whitespace characters are removed:

```
sign          ::= "+" | "-"
infinity      ::= "Infinity" | "inf"
nan           ::= "nan"
numeric_value ::= floatnumber | infinity | nan
numeric_string ::= [sign] numeric_value
```


Passing an incorrect argument

```
float(2)
```

```
2.0
```

```
float('2.3')
```

```
2.3
```

```
float('hello')
```

```
<hr />-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-3-d0ce8bccc8b2> in <module>()  
<hr />-> 1 float('hi')  
ValueError: could not convert string to float: 'hello'
```

Passing valid arguments

```
def sqrt(x):  
    """Returns the square root of a number."""  
    return x ** (0.5)  
  
sqrt(4)
```

```
2.0
```

```
sqrt(10)
```

```
3.1622776601683795
```

Passing invalid arguments

```
sqrt('hello')
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-4-cfb99c64761f> in <module>()  
----> 1 sqrt('hello')  
<ipython-input-1-939b1a60b413> in sqrt(x)  
      1 def sqrt(x):  
----> 2     return x**(0.5)  
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

Errors and exceptions

- Exceptions - caught during execution
- Catch exceptions with try-except clause
 - Runs the code following try
 - If there's an exception, run the code following except

Errors and exceptions

```
def sqrt(x):  
    """Returns the square root of a number."""  
    try:  
        return x ** 0.5  
    except:  
        print('x must be an int or float')  
  
sqrt(4)
```

```
2.0
```

```
sqrt(10.0)
```

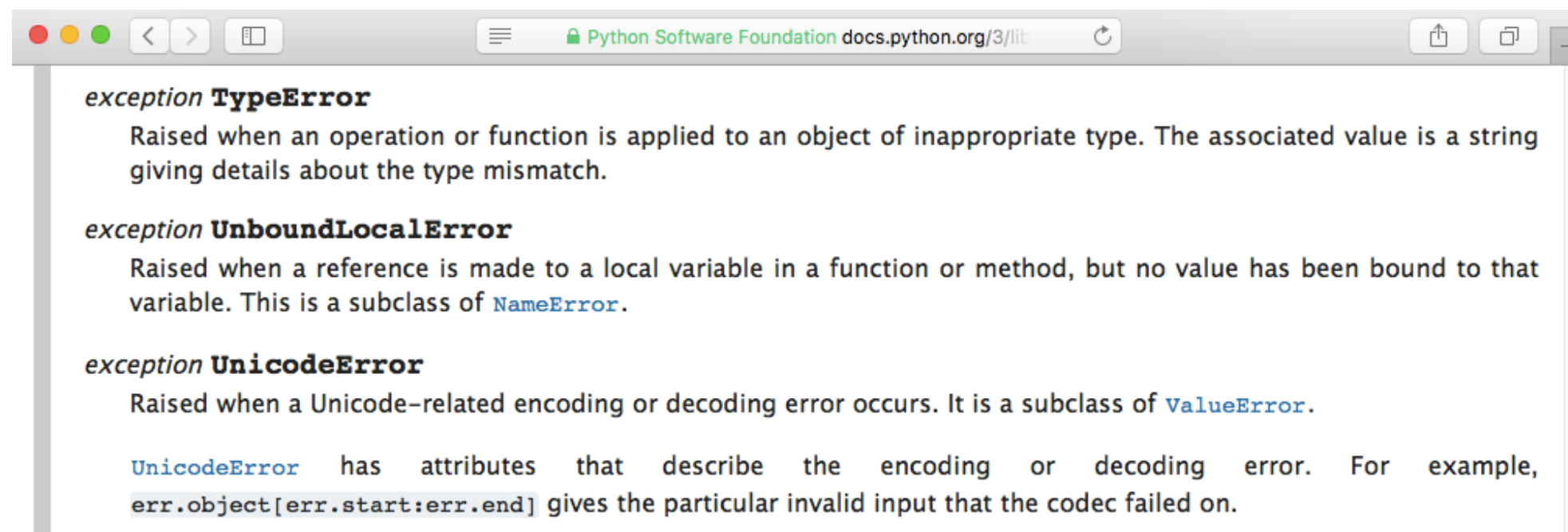
```
3.1622776601683795
```

```
sqrt('hi')
```

```
x must be an int or float
```

Errors and exceptions

```
def sqrt(x):  
    """Returns the square root of a number."""  
    try:  
        return x ** 0.5  
    except TypeError:  
        print('x must be an int or float')
```



Errors and exceptions

```
sqrt(-9)
```

```
(1.8369701987210297e-16+3j)
```

```
def sqrt(x):  
    """Returns the square root of a number."""  
    if x < 0:  
        raise ValueError('x must be non-negative')  
    try:  
        return x ** 0.5  
    except TypeError:  
        print('x must be an int or float')
```

Errors and exceptions

```
sqrt(-2)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-2-4cf32322fa95> in <module>()  
----> 1 sqrt(-2)  
<ipython-input-1-a7b8126942e3> in sqrt(x)  
      1 def sqrt(x):  
      2     if x < 0:  
----> 3         raise ValueError('x must be non-negative')  
      4     try:  
      5         return x**(0.5)  
ValueError: x must be non-negative
```


Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Bringing it all together

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

Errors and exceptions

```
def sqrt(x):  
    try:  
        return x ** 0.5  
    except:  
        print('x must be an int or float')
```

```
sqrt(4)
```

```
2.0
```

```
sqrt('hi')
```

```
x must be an int or float
```

Errors and exceptions

```
def sqrt(x):  
    if x < 0:  
        raise ValueError('x must be non-negative')  
    try:  
        return x ** 0.5  
    except TypeError:  
        print('x must be an int or float')
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Congratulations!

PYTHON DATA SCIENCE TOOLBOX (PART 1)



Hugo Bowne-Anderson
Instructor

What you've learned:

- Write functions that accept single and multiple arguments
- Write functions that return one or many values
- Use default, flexible, and keyword arguments
- Global and local scope in functions
- Write lambda functions
- Handle errors

There's more to learn!

- Create lists with list comprehensions
- Iterators - you've seen them before!
- Case studies to apply these techniques to Data Science

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)