



PYTHON DATA SCIENCE TOOLBOX I

User-defined functions



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()`

```
In [1]: x = str(5)
```

```
In [2]: print(x)  
'5'
```

```
In [3]: print(type(x))  
<class 'str'>
```



Defining a function

```
In [1]: def square():  
...:     new_value = 4 ** 2  
...:     print(new_value)
```

Function header

**Function body
(Indented)**

```
In [2]: square()  
16
```



Function parameters

```
In [1]: def square(value): ← parameter
...:     new_value = value ** 2
...:     print(new_value)
```

```
In [2]: square(4) ← argument
16
```

```
In [3]: square(5)
25
```



Return values from functions

- Return a value from a function using `return`

```
In [1]: def square(value):  
...:     new_value = value ** 2  
...:     return new_value
```

```
In [12]: num = square(4)
```

```
In [13]: 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 `"""`

```
In [1]: def square(value):  
...:     """Return the square of a value."""  
...:     new_value = value ** 2  
...:     return new_value
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Multiple parameters and return values



Multiple function parameters

- Accept more than 1 parameter:

```
In [1]: 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

```
In [2]: result = raise_to_power(2, 3)
```

```
In [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 ()

```
In [1]: even_nums = (2, 4, 6)
```

```
In [2]: print(type(even_nums))  
<class 'tuple'>
```



Unpacking tuples

- Unpack a tuple into several variables:

```
In [1]: even_nums = (2, 4, 6)
```

```
In [2]: a, b, c = even_nums
```

```
In [3]: print(a)  
2
```

```
In [4]: print(b)  
4
```

```
In [5]: print(c)  
6
```



Accessing tuple elements

- Access tuple elements like you do with lists:

```
In [1]: even_nums = (2, 4, 6)
```

```
In [2]: print(even_nums[1])  
4
```

```
In [3]: second_num = even_nums[1]
```

```
In [4]: print(second_num)  
4
```

- Uses zero-indexing



Returning multiple values

raise.py

```
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
```

```
In [1]: result = raise_both(2, 3)
```

```
In [2]: print(result)  
(8, 9)
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

**Bringing it all
together**



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

raise.py

```
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
```

Function header

Function body



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Congratulations!



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!



PYTHON DATA SCIENCE TOOLBOX I

**See you in the
next chapter!**



PYTHON DATA SCIENCE TOOLBOX I

Scope and user-defined functions



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)

```
In [1]: def square(value):  
...:     """Returns the square of a number."""  
...:     new_val = value ** 2  
...:     return new_val
```

```
In [2]: square(3)  
Out[2]: 9
```

```
In [3]: new_val
```

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



Global vs. local scope (2)

```
In [1]: new_val = 10
```

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

```
In [3]: square(3)  
Out[3]: 9
```

```
In [4]: new_val  
Out[4]: 10
```



Global vs. local scope (3)

```
In [1]: new_val = 10
```

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

```
In [3]: square(3)  
Out[3]: 100
```

```
In [4]: new_val = 20
```

```
In [5]: square(3)  
Out[5]: 400
```



Global vs. local scope (4)

```
In [1]: new_val = 10
```

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

```
In [3]: square(3)
```

```
Out[3]: 100
```

```
In [4]: new_val
```

```
Out[4]: 100
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Nested functions



Nested functions (1)

nested.py

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



Nested functions (2)

square3.py

```
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
```




Nested functions (3)

mod2plus5.py

```
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))
```

```
In [1]: print(mod2plus5(1, 2, 3))  
(6, 5, 6)
```



Returning functions

raise.py

```
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
```

```
In [1]: square = raise_val(2)  
In [2]: cube = raise_val(3)  
In [3]: print(square(2), cube(4))  
4 64
```



Using nonlocal

nonlocal.py

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

```
In [1]: outer()  
2  
2
```



Scopes searched

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



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Default and flexible arguments



You'll learn:

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



Add a default argument

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

```
In [2]: power(9, 2)  
Out[2]: 81
```

```
In [3]: power(9, 1)  
Out[3]: 9
```

```
In [4]: power(9)  
Out[4]: 9
```




Flexible arguments: *args (1)

add_all.py

```
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)

```
In [1]: add_all(1)
Out[1]: 1
```

```
In [2]: add_all(1, 2)
Out[2]: 3
```

```
In [3]: add_all(5, 10, 15, 20)
Out[3]: 50
```

Flexible arguments: ****kwargs**

```
In [1]: print_all(name="Hugo Bowne-Anderson", employer="DataCamp")  
name: Hugo Bowne-Anderson  
employer: DataCamp
```



Flexible arguments: ****kwargs**

kwargs.py

```
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)
```

```
In [1]: print_all(name="dumbledore", job="headmaster")  
job: headmaster  
name: dumbledore
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

**Bringing it all
together**



Next exercises:

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



Add a default argument

power.py

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

```
In [1]: power(9, 2)  
Out[1]: 81
```

```
In [2]: power(9)  
Out[2]: 9
```




Flexible arguments: *args (1)

add_all.py

```
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
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Lambda functions



Lambda functions

```
In [1]: raise_to_power = lambda x, y: x ** y
```

```
In [2]: raise_to_power(2, 3)  
Out[2]: 8
```



Anonymous functions

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

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

```
In [2]: square_all = map(lambda num: num ** 2, nums)
```

```
In [3]: print(square_all)
<map object at 0x103e065c0>
```

```
In [4]: print(list(square_all))
[2304, 36, 81, 441, 1]
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Introduction to error handling



The float() function

The screenshot shows a web browser window with the address bar displaying `docs.python.org/3/`. The page content is the documentation for the `float()` function. The title `class float([x])` is highlighted in yellow. Below it, the description states: "Return a floating point number constructed from a number or string x." A paragraph follows, explaining the rules for string arguments: they must contain a decimal number, optionally preceded by a sign ('+' or '-') or embedded in whitespace. It also mentions that the argument can be a string representing NaN (not-a-number) or positive/negative infinity. The input must conform to a specific grammar after removing leading and trailing whitespace. This grammar is listed in a green box at the bottom of the screenshot.

```
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

```
In [1]: float(2)
Out[1]: 2.0
```

```
In [2]: float('2.3')
Out[2]: 2.3
```

```
In [3]: float('hello')
```

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



Passing valid arguments

```
In [1]: def sqrt(x):  
...:     """Returns the square root of a number."""  
...:     return x ** (0.5)
```

```
In [2]: sqrt(4)
```

```
Out[2]: 2.0
```

```
In [3]: sqrt(10)
```

```
Out[3]: 3.1622776601683795
```



Passing invalid arguments

```
In [4]: 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

```
In [1]: def sqrt(x):  
...:     """Returns the square root of a number."""  
...:     try:  
...:         return x ** 0.5  
...:     except:  
...:         print('x must be an int or float')
```

```
In [2]: sqrt(4)  
Out[2]: 2.0
```

```
In [3]: sqrt(10.0)  
Out[3]: 3.1622776601683795
```

```
In [4]: sqrt('hi')  
x must be an int or float
```



Errors and exceptions

```
In [1]: 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')
```

The screenshot shows a web browser window with the address bar displaying `docs.python.org/3/lib`. The page content lists three exceptions:

- exception `TypeError`**
Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
- exception `UnboundLocalError`**
Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.
- exception `UnicodeError`**
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

Below the list, a paragraph states: `UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.



Errors and exceptions

```
In [2]: sqrt(-9)
Out[2]: (1.8369701987210297e-16+3j)
```

```
In [3]: 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

```
In [4]: 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
```




PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

**Bringing it all
together**



Errors and exceptions

sqrt.py

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

```
In [1]: sqrt(4)
```

```
Out[1]: 2.0
```

```
In [2]: sqrt('hi')
```

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



Errors and exceptions

sqrt.py

```
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')
```



PYTHON DATA SCIENCE TOOLBOX I

Let's practice!



PYTHON DATA SCIENCE TOOLBOX I

Congratulations!



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



PYTHON DATA SCIENCE TOOLBOX I

Congratulations!