WELLCOME

PYTHON FOR DATA SCIENCE AND AI

YA MANON

$$f(x)$$

You can have data without information but you cannot have information without data.

-Daniel Keys Maran

FUNCTIONS

In this section we'll learn to write custom **functions** to boost efficiency, import external functions stored in modules or packages, and write comprehensions

TOPICS WE'LL COVER:

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

GOALS FOR THIS SECTION:

- Identify the key components of Python functions
- Define custom functions and manage variable scope
- Practice importing packages, which are external collections of functions
- Learn how to write lambda functions and apply functions with map()
- Learn how to write comprehensions, powerful expressions used to create iterables

FUNCTIONS

- Function Components
- Defining Functions
- Variable Scope
- Modules
- Packages
- Lambda Functions
- Comprehensions

Functions are reusable blocks of code that perform specific tasks when called.

- They take in data, perform a set of actions, and return a result

Input

Inputs to a function are known as **arguments**

Data, is some form, is usually one of the arguments passed into a function

Other arguments might be options for processing the data or formatting the output

Function

Functions are blocks of code that perform a specific task

When you use a function, it is known as a **function call**, or **calling a function**

The function itself is stored somewhere else, and you are 'calling' it by using its name

Output

Outputs are **values** returned by a function

Side Effects

Side effects are changes or actions made by the function, other than the output

Functions help **boost efficiency** as programmers immensely!

THE ANATOMY OF MAX

The built-in **max()**function works the same way *(as do others like len(), sum(), etc.)*

```
In [5]: max?

Docstring:
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
```

The documentation provides details on the input, function, and output for max()

Input

Possible arguments:

- A single iterable
- Multiple objects

Function

Identifies the largest value in the argument(s) provided

Output

The largest value in the argument(s)

Side Effects

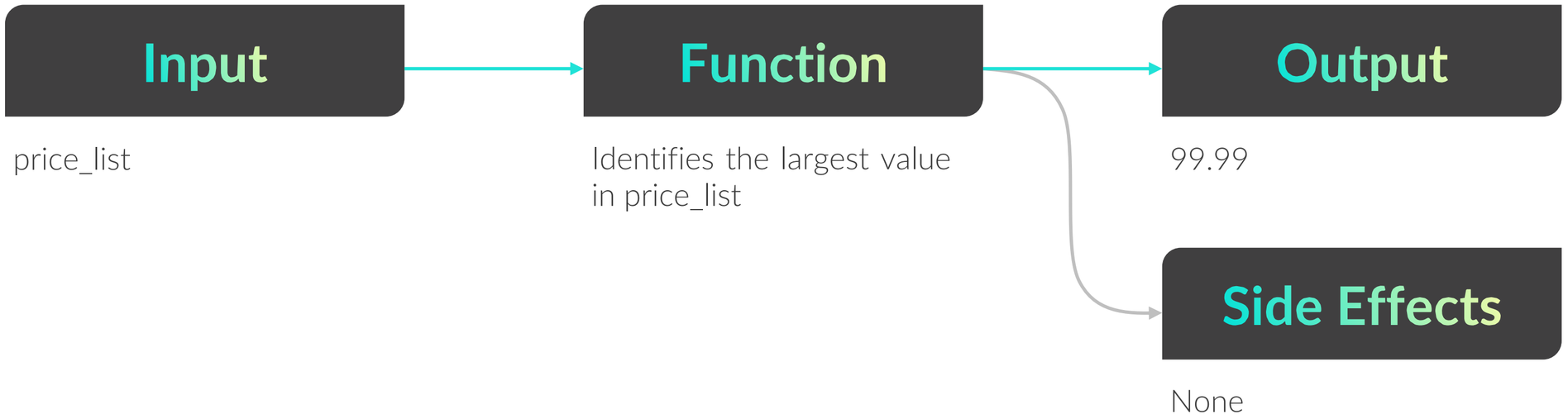
None

THE ANATOMY OF MAX

The built-in **max()**function works the same way *(as do others like len(), sum(), etc.)*

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]

max(price_list)
```



DEFINING A FUNCTION


```
def function_name(arguments):
```

Indicates you're defining a new function

An intuitive name to call you function by

Examples:

- *avg*
- *usd_converter*

Variable names for the function inputs, separated by commas

Functions have the same **naming rules** and best practices as variables –use 'snake_case'!

DEFINING A FUNCTION


```
def function_name(arguments):
    do this
```

Code block for the function that uses the arguments to perform a specific task

DEFINING A FUNCTION


```
def function_name(arguments):
    do this
    return output
```

Ends the function
(without it, the function returns None)

Values to return
(usually variables)

DEFINING A FUNCTION

EXAMPLE *Defining a function that concatenates two words, separated by a space*

```
def concatenator(string1, string2):
    combined_string = string1 + ' ' + string2
    return combined_string
```

*Here we're defining a function called **concatenator**, which accepts two arguments, combines them with a space , and returns the result*

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

When we call this function with two string arguments, the combined string is returned

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

Note that we don't need a code block before return, here we're combining strings in the return statement

WHEN TO DEFINE A FUNCTION?

You should take time to **define a function** when:

- ü You find yourself copying & pasting a block of code repeatedly
- ü You know you will use a block of code again in the future
- ü You want to share a useful piece of logic with others
- ü You want to make a complex program more readable

There are no hard and fast rules, but in data analysis you'll often have a similar workflow for different projects –by taking the time to package pieces of your data cleaning or analysis workflow into functions, you are saving your future self (and your colleagues') time

THE DOCSTRING

You can create a **docstring** for your function

- This is used to embed text describing its arguments and the actions it performs

*Use **triple quotes** inside the function to create its docstring*

```
def concatenator(string1, string2):
    """combines two strings, separated with a space

    Args:
        string1 (str): string to put before space
        string2 (str): string to put after space

    Returns:
        str: string1 and string2 separated by a space
    """
    return string1 + " " + string2
```

*Use '**?**' to retrieve the docstring, just like with built-in functions*

```
concatenator?
```

```
Signature: concatenator(string1, string2)
Docstring:
combines two strings, separated with a space

Args:
    string1 (str): string to put before space
    string2 (str): string to put after space

Returns:
    str: string1 and string2 separated by a space
File:      /var/folders/f8/075hbnj13wb0f9yzh9k4nyz00000
gn/T/ipykernel_21060/3833548733.py
Type:      function
```

PRO TIP: Take time to create a docstring for your function, especially if you plan to share it with oth
What's the point in creating reusable code if you need to read all of it to understand how to use it?

ARGUMENT TYPES

There are several **types of arguments** that can be passed on to a function:

- **Positional** arguments are passed in the order they were defined in the function
- **Keyword** arguments are passed in any order by using the argument's name
- **Default** arguments pass a preset value if nothing is passed in the function call
- ***args** arguments pass any number of positional arguments as tuples
- ****kwargs** arguments pass any number of keyword arguments as dictionaries

ARGUMENT TYPES

Positional arguments are passed in the order they were defined in the function

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

The first value passed in the function will be string1, and the second will be string2

```
concatenator('World!', 'Hello')
```

```
'World! Hello'
```

Therefore, changing the order of the inputs changes the output

ARGUMENT TYPES

Keyword arguments are passed in any order by using the argument's name

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

```
concatenator(string2='World!', string1='Hello')
```

```
'Hello World!'
```

By specifying the value to pass for each argument, the order no longer matters

```
concatenator(string2='World!', 'Hello')
```

```
SyntaxError: positional argument follows keyword argument
```

*Keyword arguments **cannot** be followed by positional arguments*

```
concatenator('Hello', string2='World!')
```

```
'Hello World!'
```

*Positional arguments **can**be followed by keyword arguments*

(the first argument is typically reserved for primary input data)

ARGUMENT TYPES

Default arguments pass a preset value if nothing is passed in the function call

```
def concatenator(string1, string2='World!'):
    return string1 + ' ' + string2
```

Assign a default value by using '=' when defining the function

```
concatenator('Hola')
```

```
'Hola World!'
```

Since a single argument was passed, the second argument defaults to 'World!'

```
concatenator('Hola', 'Mundo!')
```

```
'Hola Mundo!'
```

By specifying a second argument, the default value is no longer used

```
def concatenator(string1='Hello', string2):
    return string1 + ' ' + string2
```

```
SyntaxError: non-default argument follows default argument
```

Default arguments must come after arguments without default values

ARGUMENT TYPES

***args** arguments pass any number of positional arguments as tuples

```
def concatenator(*args):
    new_string = ''
    for arg in args:
        new_string += (arg + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
'Hello world! How are you?'
```

Using '' before the argument name allows users to enter any number of strings for the function to concatenate*

Since the arguments are passed as a tuple, we can loop through them or unpack them

```
def concatenator(*words):
    new_string = ''
    for word in words:
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!')
```

```
'Hello world!'
```

It's not necessary to use 'args' as long as the asterisk is there

Here we're using 'words' as the argument name, and only passing through two words

ARGUMENT TYPES

**kwargs arguments pass any number of keyword arguments as dictionaries

```
def concatenator(**words):
    new_string = ''
    for word in words.values():
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator(a='Hello', b ='there!',
             c="What's", d='up?')
```

```
"Hello there! What's up?"
```

*Using '**' before the argument name allows users to enter any number of keyword arguments for the function to concatenate*

Note that since the arguments are passed as dictionaries, you need to use the .values() method to loop through them

TIP: Use **kwargs arguments to unpack dictionaries and pass them as keyword arguments

```
def exponentiator(constant, base, exponent):
    return constant * (base**exponent)
```

The exponentiator function has three arguments: constant, base, and exponent

```
param_dict = {'constant': 2, 'base': 3, 'exponent': 2}

exponentiator(**param_dict)
```

```
18
```

Note that the dictionary keys in 'param_dict' match the argument names for the function

*By using '**' to pass the dictionary to the function, the dictionary is unpacked, and the value for each key is mapped to the corresponding argument*

RETURN VALUES

Functions can **return multiple values**

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

The values to return must be separated by commas

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
('Hello world! How are you?', 'you?')
```

This returns a tuple of the specified values

```
sentence, last_word = concatenator('Hello', 'world!', 'How', 'are', 'you?')

print(sentence)
print(last_word)
```

```
Hello world! How are you?
you?
```

You can unpack the tuple into variables during the function call

The variable 'sentence' is assigned to the first element returned in the tuple, so if the order was switched, it would store 'you?'

RETURN VALUES

Functions can return multiple values as **other types of iterables** as well

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return [sentence.rstrip(), last_word]
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
['Hello world! How are you?', 'you?']
```

Wrap the comma-separated return values in square brackets to return them inside a list

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return {sentence.rstrip(): last_word}

concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
{'Hello world! How are you?': 'you?'}
```

Or use dictionary notation to create a dictionary (this could be useful as input for another function!)

VARIABLE SCOPE

The **variable scope** is the region of the code where the variable was assigned

1. **Local scope** –variables created inside of a function
    - *These cannot be referenced outside of the function*

2. **Global scope** –variables created outside of functions
    - *These can be referenced both inside and outside of functions*

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

```
Hello world! How are you?
```

Since the variable 'sentence' is assigned inside of the concatenator function, it has local scope

Trying to print this variable outside of the function will then return a NameError

CHANGING VARIABLE SCOPE

You can **change variable scope** by using the *global*keyword

```
def concatenator(*words):
    global sentence
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

```
Hello world! How are you?
```

By declaring the variable 'sentence' as global, it is now recognized outside of the function it was defined in

CHANGING VARIABLE SCOPE

You can **change variable scope** by using the *global*keyword

```
def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    global sentence
    return sentence.rstrip(), last_word

concatenator('Hello', 'world!', 'How', 'are', 'you?')
print(sentence)
```

```
SyntaxError: name 'sentence' is assigned to before global declaration
```

*Note that the variable must be declared as global **before it is assigned a value**, or you will receive a SyntaxError*

PRO TIP: While it might be tempting, declaring global variables within a function is considered bad practice in most cases –imagine if you borrowed this code and it overwrote an important variable! Instead, use 'return' to deliver the values you want and assign them to local variables

CREATING MODULES

To save your functions, **create a module**in Jupyterby using the %%writefile magic command and the .py extension

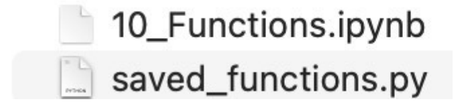
```
%%writefile saved_functions.py

def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word
```

This creates a Python modulethat you can import functions from

- *Follow **%%writefile** with the name of the file and the **.py**extension*
- *By default, the .py file is stored in the same folder as the notebook*
- *You can share functions easily by sending this file to a friend or colleague!*

```
Writing saved_functions.py
```




```
%%writefile saved_functions.py

def concatenator(*words):
    sentence = ''
    for word in words:
        sentence += word + ' '
    last_word = words[-1]
    return sentence.rstrip(), last_word


def multiplier(num1, num2):
    return num1 * num2
```

```
Overwriting saved_functions.py
```

Multiple functions can be saved to the same module

IMPORTING MODULES

To **import saved functions**, you can either import the entire module or import specific functions from the module

```
import saved_functions

saved_functions.concatenator('Hello', 'world!')
```

```
('Hello world! ', 'world!')
```

```
saved_functions.multiplier(5, 10)
```

```
50
```

import module
reads in external Python modules

*If you import the entire module, you need to reference it when calling its functions, in the form of **module.function()***

```
from saved_functions import concatenator, multiplier

concatenator('Hello', 'world!')
```

```
('Hello world! ', 'world!')
```

```
multiplier(5, 10)
```

```
50
```

frommodule importfunction
imports specific functions from modules

By importing specific functions, you don't need to reference the entire module name when calling a function

⚠ This method can lead to **naming conflicts** if another object has the same name

IMPORTING EXTERNAL FUNCTIONS

The same method used to import your own modules can be used to import external modules, packages, and libraries

- **Modules** are individual .py files
- **Packages** are collections of modules
- **Libraries** are collections of packages *(library and package are often used interchangeably)*

```
dir(saved_functions)
```

Use the dir() function to view the contents for any of the above

- *This is showing the directory for the saved_functions module*
- *Modules have many attributes associated with them that the functions will follow*
- *'__file__' is the name of the .py file*

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'concatenator',
 'multiplier']
```

These are the two functions inside the module

IMPORTING EXTERNAL FUNCTIONS

EXAMPLE | *Importing the math package, which contains handy mathematical functions*

```
import math

dir(math)
```

```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
```

This is arc cosine

This package has a LOTof functions
(too many for a single page!)

What does math.sqrt return?

```
math.sqrt(81)
```

```
9.0
```

It's helpful to look up the official documentation, which will usually give a helpful overview of the package and its contents

docs.python.org/3/library/math.html

TIP: Import packages with an alias using the **as** keyword –this saves keystrokes while still explicitly referencing the package to help avoid naming conflicts

```
import math as m

m.sqrt(81)
```

```
9.0
```

PRO TIP:NAMING CONFLICTS

Importing external functions can lead to **naming conflicts**

```
def sqrt(number):
    return f'The square root of {number} is its square root.'

sqrt(10)
```

```
'The square root of 10 is its square root.'
```

```
from math import sqrt

sqrt(10)
```

```
3.1622776601683795
```

In this case we have a sqrt function we defined ourselves, and another that we imported

In scenarios like these, only the ***most recently created*** *object with a given name will be recognized*

```
def sqrt(number):
    return f'The square root of {number} is its square root.'
```

```
import math as m

print(m.sqrt(10))
print(sqrt(10))
```

```
3.1622776601683795
The square root of 10 is its square root.
```

To avoid conflicts, refer to the package name or an alias with imported functions

INSTALLING PACKAGES

While many come preinstalled, you may need to **install a package** yourself

'!' sends the following code to your operating system's shell

(Yes, this means you can use the command line via Jupyter)

```
!conda install openpyxl
```

The name of the package to install

***conda**is a package manager for Python*

***install**tells Python to install the following package*

DO NOT install packages like this

It can lead to installing packages in the wrong instance of Python

```
import sys
!conda install --yes --prefix {sys.prefix} openpyxl
```

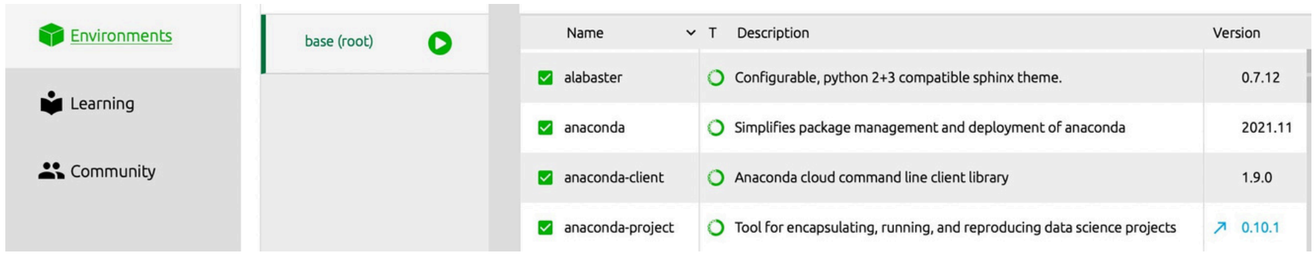
Adding this extra code snippet ensures the package gets installed correctly

PRO TIP:MANAGING PACKAGES

Use Anaconda to **review all installed packages** and their versions

```
!conda list
notebook                  6.4.8           py39hecd8cb5_0
numpy                     1.21.2          py39h4b4dc7a_0
numpy-base                1.21.2          py39he0bd621_0
openpyxl                  3.0.9             pyhd3eb1b0_0
```

condalist*returns a list of installed packages and versions*



*The **environments tab**in Anaconda Navigator also shows the installed packages and versions*

```
import sys

!conda install --yes --prefix {sys.prefix} openpyxl
```

```
!conda install --yes --prefix {sys.prefix} package_name=1.2.3
```

conda install** updates a package to its latest version; to install a specific version, use **package_name=<version>

You may need to install an older version of a package so it is compatible with another package you are working with

ESSENTIAL PACKAGES FOR ANALYTICS

Python has incredible packages for **data analysis** beyond math and openpyxl Here are some to explore further:

NumPy

matplotlib

pandas

plotly

MAP

The **map()** functionis an efficient way to apply a function to all items in an iterable

- `map(function, iterable)`

```
def currency_formatter(number):
    return '$' + str(number)
```

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]

map(currency_formatter, price_list)
```

```
<map at 0x7fa430944d90>
```

*The map function returns a **map object** – which saves memory until we've told Python what to do with the object*

```
list(map(currency_formatter, price_list))
```

```
['$5.99', '$19.99', '$24.99', '$0', '$74.99', '$99.99']
```

You can convert the map object into a list or other iterable

LAMBDA FUNCTIONS

Lambda functions are single line, anonymous functions that are only used briefly

- `lambda arguments: expression`

```
(lambda x: x**2) (3)
```

```
9
```

Lambda functions can be calledon a single value, but typically aren't used for this

```
(lambda x: x**2, x**3)(3)
```

```
NameError
```

They cannot have multiple outputs or expressions

```
(lambda x, y: x * y if y > 5 else x / y)(6, 5)
```

```
1.2
```

They can take multiple arguments and leverage conditional logic

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88

converted = map(lambda x: round(x * exchange_rate, 2), price_list)
list(converted)
```

```
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

They are usually leveraged in combination with a function like map() or in a comprehension.

PRO TIP: COMPREHENSIONS

Comprehensions can generate sequences from other sequences

Syntax:

```
new_list = [expression for member in other_iterable (if condition)]
```

EXAMPLE *Creating a list of Euro prices from USD prices*

Before, *you needed a for loop to create the new list*

```
usd_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]
exchange_rate = 0.88
euro_list = []

for price in usd_list:
    euro_list.append(round(price * exchange_rate, 2))

euro_list
```

```
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

→ ***Now***, *you can use comprehensions to do this with a single line of code*

```
exchange_rate = 0.88
usd_list = [5.99, 9.99, 19.99, 24.99, 99.99]

euro_list = [round(price * exchange_rate, 2) for price in usd_list]
euro_list
```

```
[5.27, 8.79, 17.59, 21.99, 87.99]
```

```
euro_list = [round(price * exchange_rate, 2) for price in usd_list if price < 10]
euro_list
```

```
[5.27, 0.0]
```

You can even leverage conditional logic to create very powerful expressions!

PRO TIP: DICTIONARY COMPREHENSIONS

Comprehensionscan also **create dictionaries** from other iterables

Syntax:

```
new_dict = {key: value for key, value in other_iterable(if condition)}
```

These can be expressions (including function calls!)

EXAMPLE *Creating a dictionary of inventory costs per item (stock quantity * price)*

```
items = ['skis', 'snowboard', 'goggles', 'boots']
status = [[5, 249.99], [0, 219.99], [0, 99.99], [12, 79.99]]

inventory_costs = {k: round(v[0] * v[1], 2) for k, v in zip(items, status)}

inventory_costs
```

```
{'skis': 1249.95, 'snowboard': 0.0, 'goggles': 0.0, 'boots': 959.88}
```

*This is creating a dictionary by using **items**as keys, and the product of **status[0]** and **status[1]**as values*

zip() is being used to stitch the two lists together into a single iterable

```
inventory_costs = {
    k: round(v[0] * v[1], 2) for k, v in zip(items, status) if v[0] > 0
}

inventory_costs
```

```
{'skis': 1249.95, 'boots': 959.88}
```

You can still use conditional logic!

TIP: COMPREHENSIONS VS MAP

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

Note that exchange_rate is a default argument equal to .88

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]

[currency_converter(price) for price in price_list]
```

```
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

Here, in both the comprehension and map(), price_list is being passed as a positional argument to the currency_converter function, and the default exchange_rate value is applied

```
list(map(currency_converter, price_list))
```

```
[5.27, 17.59, 21.99, 0.0, 65.99, 87.99]
```

But what if I want to change the exchange rate?

PRO TIP: COMPREHENSIONS VS MAP

Comprehensions, like map(), can apply functions to an entire sequence

```
def currency_converter(price, exchange_rate=.88):
    return round(float(price) * exchange_rate, 2)
```

Note that exchange_rate is a default argument equal to .88

```
price_list = [5.99, 19.99, 24.99, 0, 74.99, 99.99]

[currency_converter(price, exchange_rate=.85) for price in price_list]
```

In the comprehension, the exchange_rate argument is easy to specify

```
import functools
list(map(functools.partial(currency_converter, exchange_rate=.85), price_list))
```

```
[5.09, 16.99, 21.24, 0.0, 63.74, 84.99]
```

To do so with map(), you need to create a partial function (outside the course scope)

In general, both methods provide an efficient way to apply a function to a list –Comprehensions are usually preferred, as they are more efficient and highly readable, but there will be instances when using map with lambda is preferred ***(like manipulating a Pandas Column)***

You may prefer not to use comprehensions for particularly complex logic ***(nested loops, multiple sequences, lots of conditions)***, but for most use cases comprehensions are a best practice for creating new sequences based off others.

KEY TAKEAWAYS

Functions are reusable blocks of code that make us more efficient analysts

- *If you find yourself writing the same code over and over again, consider making it a function*

External**packages** &**libraries**provide access to functions developed by others

- *Most data analysts work with features from packages like pandas and matplotlib regularly*

Map() applies a function to an iterablewithout the use of a for loop

- *This is often paired with lambda functions for handy one-off procedures*

Comprehensions cancreate sequences and dictionaries from other iterables

- *They are more efficient than map() at applying a function to a list, and allow you to specify different arguments*