topics:

1:numpy

2:Pandas

3:MatPlotLib:

4:misc:

5:packaging:

6:invoke:

7:flask:

8: conda related:

doubts: why corresponding environment is not shown as active even if we launch that environment from wsl

-You can manually specify the path to the conda executable to use for activation (version 4.4+). To do so, open the Command Palette (Ctrl+Shift+P) and enter Preferences: Open User Settings. Then set python.condaPath, which is in the Python extension section of User Settings, with the appropriate path.

-we can have different environments in conda, and each environment can have different packages installed.

-to activate a particular base we have to use - conda activate <environment name>

-similarly to deactivate a particular base we have to use - conda deactivate <environment name>

-to remove existing package:

conda env remove -n <name of package >

-to see the list of info's type:

conda info --envs

(snowflakes) saivinil\_pratap@TIGER02143:~$ conda info --envs

# conda environments:

#

base /home/saivinil\_pratap/miniconda3

snowflakes \* /home/saivinil\_pratap/miniconda3/envs/snowflakes

the \* shows that the snowflakes is the current active environment,alternatively you can confirm this by seeing the name that is in the brackets above when we try to execute the command "conda info --envs"

-if you want to move back to base environment, you can simply type-

conda activate

this will move the environment to default environment which is base.

-when you create a new environment conda uses the same python version which is used to download and install anaconda, if you want to create a new environment with different python version, then you have to use the following command (conda env create or create env):

conda create --name <environment name> python=<version>

ex: conda create --name MLproj-env python=3.7

-to delete conda environment:

conda env remove -n <env name>

-if you want to search for a package in ANACONDA REPOSITORY

ex: conda search \*eauti\* (it supports wild card entries and it will check the repository and lists the package that has "eauti" inside it )

-pipenv run black <python file path>

black is a package that will try to format code according to the python standards

this command will try to make the code structured by making considerable changes that will make the code look neat eloquent,elagant and easy to understand (to do this you have to install pipenv and black)

-pipenv run isort <python file path>

isort is a package in python,isort helps to sort and format imports in your Python code. Black also formats imports, but in a different way from isort’s defaults which leads to conflicting changes.

-flake8:

it is a code linter in python, a code linter is basically a program that will check your code and gives feedback.it can detect issues in the program and suggest solutions to resolve them. you can run the linter anytime to let the code have good standards

-if you want to debug python code in visual studio: place the cursor on the line where you want to add a debug point and press f9

-plots dont work with wsl (version 1) configuration of vscode, but you can get plots by right clicking on them and then running it in interactive mode.

-when you are using virtual environments in vs code then, in the terminal you have to activate corresponding environment and then press ctrl+shigt+p and select the same interpreter

-you have used some commands or functions and terminated session, if you want to get them back you can use the reverse-i-search which can be implemented by shortcut ctrl+r is useful in linux related systems. commands/functions that are used before and after terminating the session can be recalled

-setting environment variables:

1. import os

os.environ['GOOGLE\_APPLICATION\_CREDENTIALS'] = '/home/saivinil\_pratap/mlops-with-vertex-ai-v2/key-by-dinesh-tiger-mle-e5a6296c4fb4.json'

1. !export GOOGLE\_APPLICATION\_CREDENTIALS=\home\saivinil\_pratap\mlops-with-vertex-ai-v2\tiger-mle-8c54fa5ce18f.json
2. os.environ["GOOGLE\_APPLICATION\_CREDENTIALS"] #to check wether it is assigned or not

-When you are working with a virtual environment and you had installed specific packages to cover all the needs and you're ready to deploy the application to other computers, you can create a requirements.txt file with the command pip freeze > requirements.txt (pip3 on macOS/Linux). The requirements file describes the packages you've installed in your virtual environment. With only this file, you or other developers can restore those packages using pip install -r requirements.txt (or, again, pip3 on macOS/Linux). By using a requirements file, you need not commit the virtual environment itself to source control.

-If you create a new conda environment while VS Code is running, use the Reload Window command to refresh the environment list shown with Python: Select Interpreter; otherwise you may not see the environment there. It might take a short time to appear; if you don't see it at first, wait 15 seconds then try using the command again.

-if you want to launch vs code for a particular folder while you are using WSL, then just navigate to the folder in linux and then give the command "code .", then you can open that folder in vs code without any hassle, if you just want to load vs code without any directory (cwd in linux) being prioritized just enter "code"

-if you are having a lot of text(as command which you dont want to execute) in terminal and if you want to remove, press ctrl+u

-if you are working from vs code using python installation and you want to activate the command code, then you have to type "shell command:code install code command in the path" in the command palette and then select it, so from now you will have the option to type "code" from the cmd(command prompt) and then launch it in vscode

-Select and activate an environment#

By default, the Python extension looks for and uses the first Python interpreter it finds in the system path. To select a specific environment, use the Python: Select Interpreter command from the Command Palette (Ctrl+Shift+P).

The Python: Select Interpreter command displays a list of available global environments, conda environments, and virtual environments. (See the Where the extension looks for environments section for details, including the distinctions between these types of environments.) The following image, for example, shows several Anaconda and CPython installations along with a conda environment and a virtual environment (env) that's located within the workspace folder:

-we can also open ipynb files, all we have to do is try opening the folder that already has ipynb files and it will automatically configure that and open it as a notebook

-in vscode-to shift control from editor to terminal, you can create a shortcut by going to command palette(ctrl+shift+p) and search "focus terminal" and add custom shortcut, similarly to shift focus back to editor- Ctrl + E , Enter.

else you can simply press ctrl+~ to minimize or activate terminal(current one if exists, or a new one if it does not exists)

-if you want to navigate within the filesystem and then press ctrl+e and then type the starting words of the file and vscode will start giving suggestions, press enter (when you see the file you want to navigate to) and you just will be moved to that file. Now you want to go to a particular variable type "@" (after pressing ctrl+e because this will put preference of variables first and methods later) and then start typing the variable name, it will try to fetch the results, if you want to search for methods type "@:" (after pressing ctrl+e,then it will prefer listing methods first, and when you start typing some letters it will try to filter accordingly)

-code completion is a useful feature and at any point, if you have to use suggestions to perform autocompletion at anypoint, simply press ctrl+space (if it does not work, check for the shortcut "suggest" and it will be understandable from that point,to see what should be done)

-suppose you have a method/ variable declared somewhere else and you are using it in the current line a and you want to edit the variable/method, then by pressing alt+f12 it is feasible to edit the method/variable as it will show a popup of its declaration in current line

-suppose you want to see all the function definitions, go to function declaration and click on anywhere in the name of the function, press shift+f12, in the popup that appears, in the right side you can see the declarations of the function, and clicking on any of it will move you to the that part of file

-if you want to test the methods, install pytest and then it will search for python files that start with test, and in the files that start with test and then it will automatically display them in testing sidebar, you can run them, debug them all or one by one

-pipx

we might have different packages installed in different environments, and if you want some package which is commonly used by all environments (maybe a package which deals with code formatting- black8 or codelinter which is used for linitng), we can install them using pipx so that the packages are installed in an isolated environment which can be used for further need.

-a python package must have \_\_init\_\_.py file(mostly it is empty, it is called a package only if it has \_\_init\_\_.py file), It is a collection of python modules. a python module is a single python file(with code no lengthier than 500 lines of code including comments). The different modules in the package are imported in a similar manner as plain modules, but with a special behavior for the \_\_init\_\_.py file, which is used to gather all PACKAGE-WIDE DEFINITIONS.

A commonly seen issue is adding too much code to \_\_init\_\_.py files. When the project complexity grows, there may be sub-packages and sub-sub-packages in a deep directory structure. In this case, importing a single item from a sub-sub-package will require executing all \_\_init\_\_.py files met while traversing the tree.

Leaving an \_\_init\_\_.py file empty is considered normal and even good practice, if the package’s modules and sub-packages do not need to share any code.

-package names are lower case with an underscore to seperate words, and are always singular

-module names are lowercase with an underscore to seperate words

-Lines must not exceed 79 characters in length

-imports are always placed after module comments and doc string(which are placed at the starting of the file), and placed above the module globals and constants

Imports should be grouped in the following order:

standard library imports

related third party imports

local application/library specific imports

You should put a blank line between each group of imports

-instead of using from <packagename> import \*, we can use from <packagename> import <module1>,<module2>... and so on as the first declaration gets all the methods as identifiers which is not a expected behaviour, so the second mode of declaration will give the better approach because we will list only the required identifiers.

-using comments to let the flow understand is a good approach, but comments that are not up to date with the code changes is worse than having no comments at all, updating comments along with the code is the only better approach you can have with comments, comments should be complete sentences, it should begin with a capital letter, unless starts with an identifiers.

-BLOCK COMMENTS: comments that consist of one or more paragraph, each paragraph is built with complete sentences, each statement ends with a period, and after each period you should use double spaces before you start another sentence.

-The first line of a docstring should be a one line summary. As noted in the example below. Follow that line with a blank line, and then continue with a description or examples as appropriate. Examples are called out on a new line, indented an additional four spaces, and begin with >>>. This convention will produce nicely formated examples in the resulting html pages, so do not hesitate to use it. Conventions for writing good docstrings are found in PEP 257

-Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line.

-An inline comment is a comment on the same line as a statement. Inline comments should be used SPARINGLY. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space. Remember to use SPARINGLY, although sometimes, they can be useful

ex:s = s + 1 # Compensate for the border.

-class definitions must contain doc string, class name should be CamelCase

-'Special' methods (ie. those methods starting and ending in double underscores) should usually be grouped at the start of the class definition. The exception is when the class is implementing the interface of a built-in type such as 'list' or 'dict'. In that case the special methods on the interface follow the rules listed below.

-a protected variable or a non public variable starts with single underscore and is intended for only use of current class and the classes that derive the current class

-a private attribute is preceeded by a double underscore, they can be used only inside the class.

-functions that does not return also should be delimited by a return (i.e, a function should have an empty return at the end, even if it does not return any value)

Methods should be grouped and ordered as follows:- 1) 'object interface (ie. special methods). 2) Methods offered on inherited or delegated interfaces of the class. 3) Methods offered on the primary public interface of the class. 4) Non-public methods 5) private methods

-A method definition must contain a docstring. Method names are lowercase with an underscore used to separate each word. The ONLY exception to this is if you are inherting from a 3rd-party class that uses a different naming convention (eg. wxPython uses CamelCase method names). In that case you must follow the same style as the 3rd-party class (assuming it is consistent of course ;^).

-coding standard makes all the developers follow set of guidelines, which in turn will formulate the code, which will let other developers easily understand the code and also let us maintain the code with enhanced effeciency

-locals() and globals() are symbol tables(which will return dictionaries) which will store the symbol tables(a data structure maintaned by compiler,which contains all necessary information about the program) which inturn will store the local variables (information related to local scope)and global variables(functions or variables which are not associated with any class or function) information

-arbitary arguments will be passed as a tuple, whereas with arbitary key word arguments the arguments are passed inside a dictionary

arbitary arguments:

def arb\_arg(arg1,\*arg2):

print(type(arg2))#it is passed as tuple here

kwargs ex:

def myFun(\*\*kwargs):

for key, value in kwargs.items():#here kwargs is a dictionary

print ("%s == %s" %(key, value))

# Driver code

myFun(first ='Geeks', mid ='for', last='Geeks')

for arbitrary arguments and arbitrary kwargs, these are to be used when there is a necessity and no other clear and easy construct is there to fulfill the need

- in python no properties are considered private, but there is a convention to have double underscore before a variable or a method to convey that it is private. a private method or variable can not be accessed outside the class,private variables and private methods are acessed by only public and protected methods inside the class.

-The main convention for private properties and implementation details is to prefix all “internals” with an underscore. If the client code breaks this rule and accesses these marked elements, any misbehavior or problems encountered if the code is modified is the responsibility of the client code.any method or property that is not intended to be used by client code should be prefixed with an underscores

-

filename = 'foobar.hi.txt'

basename, \_\_, ext = filename.split('.')

Note

Many Python style guides recommend the use of a single underscore “\_” for throwaway variables rather than the double underscore “\_\_” recommended here. The issue is that “\_” is commonly used as an alias for the gettext() function, and is also used at the interactive prompt to hold the value of the last operation. Using a double underscore instead is just as clear and almost as convenient, and eliminates the risk of accidentally interfering with either of these other use cases.

-it is often a good idea to use sets or dictionaries instead of lists in cases where:

The collection will contain a large number of items

You will be repeatedly searching for items in the collection

You do not have duplicate items.

for smaller collections or collections for which you do not perform frequent searching, it is prefered to use lists, as it is memory used to setup hash table is greater than the time saved by improved search speed

-generator expressions (http://docs.python.org/tutorial/classes.html#generator-expressions) are similar to list comprehensions, except that list comprehensions store the list values, whereas generator expressions does not, they just yield the results, it does not return value, it creates a generator object.

note the example below:

def make\_batches(items, batch\_size):

"""

>>> list(make\_batches([1, 2, 3, 4, 5], batch\_size=3))

[[1, 2, 3], [4, 5]]

"""

current\_batch = []

for item in items:

current\_batch.append(item)

if len(current\_batch) == batch\_size:

yield current\_batch

current\_batch = []

yield current\_batch

-Never remove items from a list while you are iterating through it.Use a list comprehension or generator expression

-when you are assigning a list named a to a variable b and then if you want to make some changes in the list a, those changes will also reflect in b to avoid that, you need to

1:use deepcopy method while assigning

2:use list comprehension if you are operating on all the litems in list based on some conditions

-Use the with open syntax to read from files. This will automatically close files for you, even if an exception is raised in the block.

with open(log.txt) as f:

for line in f:

print(line)

note: the above block is called with block

-when a logical line of code is longer than the expected limit, then the lines can be seperated by back slash '\', but it is FRAGILE BECAUSE ANY SPACE AFTER THE BACK SLASH WILL RAISE UNEXPECTED RESULTS.A better solution is to use parentheses around your elements. Left with an unclosed parenthesis on an end-of-line, the Python interpreter will join the next line until the parentheses are closed. The same behavior holds for curly and square braces.

ex:

my\_very\_big\_string = (

"For a long time I used to go to bed early. Sometimes, "

"when I had put out my candle, my eyes would close so quickly "

"that I had not even time to say “I’m going to sleep.”"

)

from some.deep.module.inside.a.module import (

a\_nice\_function, another\_nice\_function, yet\_another\_nice\_function)

-Spaces are the preferred indentation method

-max length of a line in python is suggested as 79, whereas flowing long blocks of text with minimum structural restrictions (doc strings and comments) line length is confined to 72 characters

-Should a line break before or after a binary operator?

method1:

# No: operators sit far away from their operands

income = (gross\_wages +

taxable\_interest +

(dividends - qualified\_dividends) -

ira\_deduction -

student\_loan\_interest)

method2:

# Yes: easy to match operators with operands

income = (gross\_wages

+ taxable\_interest

+ (dividends - qualified\_dividends)

- ira\_deduction

- student\_loan\_interest)

method 2 is prefered as it is more clear

-when to use Blank Lines?

Surround top-level function and class definitions with two blank lines.

Method definitions inside a class are surrounded by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

-Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).

Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration Use blank lines in functions, sparingly, to indicate logical sections.

-The function is stateless or a pure function; i.e. it does not have any side-effects (e.g. change some global variable, or change env in any way) and will always return the same output for a given input regardless of when the function is called.

While pure functional style wouldn’t be appropriate, some of the ideas are good to leverage. When possible, favor writing stateless/pure functions. These functions don’t depend on anything other than the input arguments. This makes them easy to reason about and test. This leads to code that is easier to maintain and robust. Lazy computation is another principle that can be used when working with large amounts of data. In Python, generators provide a good way to incorporate the idea of a lazy iterator. Functions are first-class citizens in python and can be passed as an argument to another function just like any other object. This can be leveraged to build generic functions.

A recursive datastructure is used to express the algorithm. Recursion is very frequently used in this style to break down a larger problem into smaller problems which are easily solved. Here we use a list datastructure which has the nice recursive property that a subset of the list is also a list.imagine using a REDUCE to perform sum operation on list)

Object oriented approach may sometime lead to concurrency issue or race conditions, Another way to say the same thing is to suggest using functions and procedures with as few implicit contexts and side-effects as possible. A function’s implicit context is made up of any of the global variables or items in the persistence layer that are accessed from within the function. Side-effects are the changes that a function makes to its implicit context. If a function saves or deletes data in a global variable or in the persistence layer, it is said to have a side-effect.

Carefully isolating functions with context and side-effects from functions with logic (called pure functions) allows the following benefits:

Pure functions are deterministic: given a fixed input, the output will always be the same.

Pure functions are much easier to change or replace if they need to be refactored or optimized.

Pure functions are easier to test with unit tests: There is less need for complex context setup and data cleaning afterwards.

Pure functions are easier to manipulate, decorate, and pass around.

-the procedural oriented programming paradigm in python is the natural paradigm of python

-ipython is short for interactive python, which is similar to jupyter notebook, which will help write code one line at a time.

- a group of reusable lines are clubbed in to function

a group of reusable functions(that fall in to a relatable area) are clubbed in to a module

a group of modules (that make the work related to a particular concept easier) are grouped in to package

-project structure:

main package or python script,setup.py,requirements.txt should be placed at the root repository

-site-packages- it is the target directory of manually built Python packages. When you build and install Python packages from source (using distutils, probably by executing python setup.py install), you will find the installed modules in site-packages by default

-Makefile

Location->./Makefile

Purpose->Generic management tasks.

If you look at most of my projects or any Pocoo project, you’ll notice a Makefile lying around. Why? These projects aren’t written in C… In short, make is an incredibly useful tool for defining generic tasks for your project.

1-Modules

Python modules are one of the main abstraction layers available and probably the most natural one. Abstraction layers allow separating code into parts holding related data and functionality.

To keep in line with the style guide, keep module names short, lowercase, and be sure to avoid using special symbols like the dot (.) or question mark (?). A file name like my.spam.py is the one you should avoid! Naming this way will interfere with the way Python looks for modules.(In the case of my.spam.py Python expects to find a spam.py file in a folder named my which is not the case)

the import modu statement will look for the proper file, which is modu.py in the same directory as the caller, if it exists. If it is not found, the Python interpreter will search for modu.py in the “path” recursively and raise an ImportError exception when it is not found.

When modu.py is found, the Python interpreter will execute the module in an isolated scope. Any top-level statement in modu.py will be executed, including other imports if any. Function and class definitions are stored in the module’s dictionary

In many languages, an include file directive is used by the preprocessor to take all code found in the file and ‘copy’ it into the caller’s code. It is different in Python: the included code is isolated in a module namespace, which means that you generally don’t have to worry that the included code could have unwanted effects, e.g. override an existing function with the same name.(this is why the following import statement: from modu import \*. is generally considered bad practice. Using import \* makes the code harder to read and makes dependencies less compartmentalized.)

namepsace-A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves

import scenarios:

Very bad

[...]

from modu import \*

[...]

x = sqrt(4) # Is sqrt part of modu? A builtin? Defined above?

Better

from modu import sqrt

[...]

x = sqrt(4) # sqrt may be part of modu, if not redefined in between

Best

import modu

[...]

x = modu.sqrt(4) # sqrt is visibly part of modu's namespace

-A file modu.py in the directory pack/ is imported with the statement import pack.modu. This statement will look for \_\_init\_\_.py(this code will be run when the package is imported) file in pack and execute all of its top-level statements. Then it will look for a file named pack/modu.py and execute all of its top-level statements. After these operations, any variable, function, or class defined in modu.py is available in the pack.modu namespace.

Lastly, a convenient syntax is available for importing deeply nested packages: import very.deep.module as mod. This allows you to use mod in place of the verbose repetition of very.deep.module.

-functions are first-class objects(A first class object is an entity that can be dynamically created, destroyed, passed to a function, returned as a value, and have all the rights as other variables in the programming language have.)

-Decorators:

The Python language provides a simple yet powerful syntax called ‘decorators’. A decorator is a function or a class that wraps (or decorates) a function or a method. The ‘decorated’ function or method will replace the original ‘undecorated’ function or method. Because functions are first-class objects in Python, this can be done ‘manually’, but using the @decorator syntax is clearer and thus preferred.

def foo():

# do something

def decorator(func):

# manipulate func

return func

foo = decorator(foo) # Manually decorate

@decorator

def bar():

# Do something

# bar() is decorated

-Context Managers

A context manager is a Python object that provides extra contextual information to an action. This extra information takes the form of running a callable upon initiating the context using the with statement, as well as running a callable upon completing all the code inside the with block. The most well known example of using a context manager is shown here, opening on a file:

with open('file.txt') as f:

contents = f.read()

-Python calls \_\_enter\_\_ when execution enters the context of the with statement and it's time to acquire the resource. When execution leaves the context again, Python calls \_\_exit\_\_ to free up the resource. Writing a class-based context manager isn't the only way to support the with statement in Python, observe the below example

class CustomOpen(object):

def \_\_init\_\_(self, filename):

self.file = open(filename)

def \_\_enter\_\_(self):

return self.file

def \_\_exit\_\_(self, ctx\_type, ctx\_value, ctx\_traceback):

self.file.close()

with CustomOpen('file') as f:

contents = f.read()

expl:

This is just a regular Python object with two extra methods that are used by the with statement. CustomOpen is first instantiated and then its \_\_enter\_\_ method is called and whatever \_\_enter\_\_ returns is assigned to f in the as f part of the statement. When the contents of the with block is finished executing, the \_\_exit\_\_ method is then called.

- str, int, long, bool, float, tuple are all immutable datatypes in python

-using join() is not always best. In the instances where you are creating a new string from a pre-determined number of strings, using the addition operator is actually faster. But in cases like above or in cases where you are adding to an existing string, using join() should be your preferred method.

-Modules

A python “module” is a single namespace, with a collection of values:

functions

constants

class definitions

really any old value.

A module usually corresponds to a single file: something.py

- import os

os.makedirs(<directory path>, exist\_ok=True) #this will create a directory if the path is not exist and it will not raise an exception because of exist\_ok =True, if it is false, it will raise an error.

-after creating a package,you have to navigate to the folder, fill setup.py details then run the following commandd

pip install -e .

python setup.py sdist bdist\_wheel(generates the .whl file in dist folder in current directory, this is the installable that will be used when you want to install packages)

pip install <whl file path that ends with .whl>(Install the .whl file generated in above step which will be in dist folder)

-python3 -m pip install -U <list of packages you want to install which will be seperated by a space>

example:

python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn

to install virtual environments using pip:(again, if you want virtualenv to be installed for all users on your machine, remove --user and run this command with administrator rights)

python3 -m pip install --user -U virtualenv

-- as we do virtual environments in conda, we can also do it in normal python using commands

-describe() method in pandas ignores null values i.e, if there are 30 fields and 10 fields are null's then it will take 20 in to consideration not 30 to calculate mean,count,average and so on...

-hist() method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute

code:

%matplotlib inline

import matplotlib.pyplot as plt

housing.hist(bins=50, figsize=(20,15))

plt.show()

-explanation for %matplotlib inline:

The hist() method relies on Matplotlib, which in turn relies on a user-specified graphical backend to draw on your screen. So before you can plot anything, you need to specify which backend Matplotlib should use. The simplest option is to use Jupyter’s magic command %matplotlib inline. This tells Jupyter to set up Matplotlib so it uses Jupyter’s own backend. Plots are then rendered within the notebook itself. Note that calling show() is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

-df.drop([column or list of columns])#will not remove the columns and just assigns them or returns them and columns will only be removed when you use inplace=True

-check out the notes in this link(https://docs.python.org/3/howto/argparse.html) and add points to argparse when free.

-list(df) will give the column names list

--argparse

import argparse #this module helps in having better features to pass argument

parser = argparse.ArgumentParser()

parser.add\_argument("echo", help="echo the string you use here") #help messages are displayed in detail if you use python <scriptname.py> -h (h here stands for help,) echo is the variable to which the parameter1 is assigned

parser.add\_argument("echo1", help="echo1 the string you use here")

args = parser.parse\_args()

print(args)

execution:

<scriptname.py> hi there #this for above example will be as echo='hi', echo1='there'

by default all the arguments are treated as strings, if you want to treat the parameter as int you have to explicilty specify it like below.

parser.add\_argument("num1", help="this is the first argument", type=int)

parser.add\_argument("num2", help="this is the second argument", type=int)

abc.py 4 6

-verbose in argparse

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("--verbosity", help="increase output verbosity")

args = parser.parse\_args()

if args.verbosity:

print("verbosity turned on")

And the output:

$ python3 prog.py --verbosity 1

verbosity turned on

$ python3 prog.py

$ python3 prog.py --help

usage: prog.py [-h] [--verbosity VERBOSITY]

options:

-h, --help show this help message and exit

--verbosity VERBOSITY

increase output verbosity

$ python3 prog.py --verbosity

usage: prog.py [-h] [--verbosity VERBOSITY]

prog.py: error: argument --verbosity: expected one argument

Here is what is happening:

The program is written so as to display something when --verbosity is specified and display nothing when not.

To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn’t used, the relevant variable, in this case args.verbosity, is given None as a value, which is the reason it fails the truth test of the if statement.

The help message is a bit different.

When using the --verbosity option, one must also specify some value, any value.

but for our simple program, only two values are actually useful, True or False. Let’s modify the code accordingly:

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("--verbose", help="increase output verbosity",

action="store\_true")#notice we used verbose, not verbosity like we have used in above example

#if you use parser.add\_argument("-v", "--verbose", help="increase output verbosity",

action="store\_true") instead of above line, -v can be used as shortcut while executing code you can give

python3 <script address.py> -v, it will turn verbosity on

args = parser.parse\_args()

if args.verbose:#notice we used verbose, not verbosity like we have used in above example

print("verbosity turned on")

And the output:

$ python3 prog.py --verbose #notice that we passed verbose not verbosity, if we dont pass any value along with it, it is taken as true, there by the above loop executes, if you don't pass --verbose while running it will be passed as false and it will not execute if loop above, and wont raise an error

verbosity turned on

$ python3 prog.py --verbose 1

usage: prog.py [-h] [--verbose]

prog.py: error: unrecognized arguments: 1

$ python3 prog.py --help

usage: prog.py [-h] [--verbose]

options:

-h, --help show this help message and exit

--verbose increase output verbosity

Here is what is happening:

The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, action, and give it the value "store\_true". This means that, if the option is specified, assign the value True to args.verbose. Not specifying it implies False.

It complains when you specify a value, in true spirit of what flags actually are.

Notice the different help text.

-you can combine both positional and optional arguments to have the functionality of verbosity to some good use:

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("square", type=int,

help="display a square of a given number")

parser.add\_argument("-v", "--verbose", action="store\_true",

help="increase output verbosity")

args = parser.parse\_args()

answer = args.square\*\*2

if args.verbose:

print(f"the square of {args.square} equals {answer}")

else:

print(answer)

terminal:

$ python3 prog.py

usage: prog.py [-h] [-v] square

prog.py: error: the following arguments are required: square

$ python3 prog.py 4

16

$ python3 prog.py 4 --verbose #note:here we passed 4 first and --verbose next, it worked properly

the square of 4 equals 16

$ python3 prog.py --verbose 4 #note:here we passed --verbose first and 4 next, it worked properly, the position does not matter

the square of 4 equals 16

-we can also have multiple verbosity values as well:

code:

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("square", type=int,

help="display a square of a given number")

parser.add\_argument("-v", "--verbosity", type=int,

help="increase output verbosity")

#for above line, it accepts any number as parameter alongside verbosity, $ python3 prog.py 4 -v 3 is valid(else loop also executes), $ python3 prog.py 4 -v 1000000 is also valid(else loop also executes), if you want to restrict verbosity values to a range use the below line of code

#parser.add\_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],

help="increase output verbosity") #this will only accept 0,1,2 as values, any other value will get an error, this is more interactive and is displayed when you use -h or if you throw a wrong error

#parser.add\_argument("-v", "--verbosity", action="count",

help="increase output verbosity") #when you use parameter action = count and if you want to pass verbose value as 2, then $ python3 prog.py 4 -v 2 will raise an error, you have to give $ python3 prog.py 4 -vv or

$ python3 prog.py 4 --verbosity --verbosity (as it will count the option as the verbose parameter you want to pass) #if you don’t specify the -v flag, that flag is considered to have None value.

parser.add\_argument("-v", "--verbosity", action="count", default=0,

help="increase output verbosity") #as said in above line of code, you can have None, you cant compare it with int so you can assign a default value

args = parser.parse\_args()

answer = args.square\*\*2

if args.verbosity == 2:

print(f"the square of {args.square} equals {answer}")

elif args.verbosity == 1:

print(f"{args.square}^2 == {answer}")

else:

print(answer)

-Notice that so far we’ve been using verbosity level to change the text that gets displayed. The following example instead uses verbosity level to display more text instead:

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("x", type=int, help="the base")

parser.add\_argument("y", type=int, help="the exponent")

parser.add\_argument("-v", "--verbosity", action="count", default=0)

args = parser.parse\_args()

answer = args.x\*\*args.y

if args.verbosity >= 2:

print(f"Running '{\_\_file\_\_}'")

if args.verbosity >= 1:

print(f"{args.x}^{args.y} == ", end="")

print(answer)

Output:

$ python3 prog.py 4 2

16

$ python3 prog.py 4 2 -v

4^2 == 16

$ python3 prog.py 4 2 -vv

Running 'prog.py'

4^2 == 16

-Conflicting options

So far, we have been working with two methods of an argparse.ArgumentParser instance. Let’s introduce a third one, add\_mutually\_exclusive\_group(). It allows for us to specify options that conflict with each other. Let’s also change the rest of the program so that the new functionality makes more sense: we’ll introduce the --quiet option, which will be the opposite of the --verbose one:

import argparse

parser = argparse.ArgumentParser()

group = parser.add\_mutually\_exclusive\_group()

group.add\_argument("-v", "--verbose", action="store\_true")

group.add\_argument("-q", "--quiet", action="store\_true")

parser.add\_argument("x", type=int, help="the base")

parser.add\_argument("y", type=int, help="the exponent")

args = parser.parse\_args()

answer = args.x\*\*args.y

if args.quiet:

print(answer)

elif args.verbose:

print(f"{args.x} to the power {args.y} equals {answer}")

else:

print(f"{args.x}^{args.y} == {answer}")

Our program is now simpler, and we’ve lost some functionality for the sake of demonstration. Anyways, here’s the output:

output:

$ python3 prog.py 4 2

4^2 == 16

$ python3 prog.py 4 2 -q

16

$ python3 prog.py 4 2 -v

4 to the power 2 equals 16

$ python3 prog.py 4 2 -vq

usage: prog.py [-h] [-v | -q] x y

prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose

$ python3 prog.py 4 2 -v --quiet

usage: prog.py [-h] [-v | -q] x y

prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose

That should be easy to follow. I’ve added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don’t know:

import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")#NOTE: you can add description to an argparse object, that is visible when you use -h with python3 <script.py> [-h |--help]

group = parser.add\_mutually\_exclusive\_group()

group.add\_argument("-v", "--verbose", action="store\_true")

group.add\_argument("-q", "--quiet", action="store\_true")

parser.add\_argument("x", type=int, help="the base")

parser.add\_argument("y", type=int, help="the exponent")

args = parser.parse\_args()

answer = args.x\*\*args.y

if args.quiet:

print(answer)

elif args.verbose:

print("{} to the power {} equals {}".format(args.x, args.y, answer))

else:

print("{}^{} == {}".format(args.x, args.y, answer))

Note that slight difference in the usage text. Note the [-v | -q], which tells us that we can either use -v or -q, but not both at the same time:

$ python3 prog.py --help

usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:

x the base

y the exponent

options:

-h, --help show this help message and exit

-v, --verbose

-q, --quiet

-- one python script can have more than one main function

--when you want to add an optional argument through argparse, you have to use:

parser.add\_argument("--dataset\_path", help="path of the dataset")#notice the -- before the parameter, that is what makes the argument optional

while running the script if you want to pass value, you can use:

python3 <script path> --dataset\_path 'src/data'

if you dont want to pass you can simply pass:

python3 <script path>

--imputer will take existing values in consideration while applying the strategy.

ex: l=[1,2,nan]

if we apply mean strategy using imputer, then mean(1,2) is calculated and imputed in place of nan

the list will be =[1,2,1.5]

* Running a python script with local imports from different folders(parent level, grand parent level)

let's suppose you are importing a python script named config (which is in the same folder), you can import it in 2 ways:

1: use import statement as

(let say there is a folder in current working directory(folder) named src, inside that folder you have a py file called ingest\_data)

from folder.python\_file import classname

from src.config import Config

then you have to run as

python3 -m <parent folder.python file>

ex:python3 -m src.ingest\_data

2:use import statement as (not clear example)

from <parent foldername.filename> import <class name>

then you have to run as

python3 <parent folder.> --pd.get\_dummies work in favour of text columns similar to one hot encoding

drop\_first=True this will remove one column out of n columns so that if n-1 columns are 0, then it means the removed column value will supposedly be 1

-- if you want to downgrade to a lower version of python in an environment in conda, you can use normal conda install command

conda install python=<version>

-- you have to install jupyter notebook using command

conda install jupyter

you can launch ipython in wsl by just typing the "ipython" in wsl

similarly if you want to launch jupyter notebook , just type "jupyter notebook" in wsl, and you will get one (or more) file links and one (or more) URL's

if you copy paste anyone of above url or file, it should launch wsl's jupyter notebook

conda install jupyter-lab #this will install jupyter lab which is more comfortable type of working mode

in ipython:

?- this will give shorthand for accessing this documentation and other relevant information

In [3]: len?

Signature: len(obj, /)

Docstring: Return the number of items in a container.

Type: builtin\_function\_or\_method

In [4]: len??

Signature: len(obj, /)

Docstring: Return the number of items in a container.

Type: builtin\_function\_or\_method

In [5]: l=[1,2,3]

In [6]: l.insert?

Signature: l.insert(index, object, /)

Docstring: Insert object before index.

Type: builtin\_function\_or\_method

In [7]: l?

Type: list

String form: [1, 2, 3]

Length: 3

Docstring:

Built-in mutable sequence.

??- this will give the source code for functions

If you play with this much, you'll notice that sometimes the ?? suffix doesn't display any source code: this is generally because the object in question is not implemented in Python, but in C or some other compiled extension language. If this is the case, the ?? suffix gives the same output as the ? suffix. ?len and ??len will have same output

by pressing <tab> anywhere of python code will give you the possible code options that are inline with the operations that can be performed at that point

-Beyond tab completion: wildcard matching

Tab completion is useful if you know the first few characters of the object or attribute you're looking for, but is little help if you'd like to match characters at the middle or end of the word. For this use-case, IPython provides a means of wildcard matching for names using the \* character.

For example, WE CAN USE THIS TO LIST EVERY OBJECT IN THE NAMESPACE that ends with Warning:

In [10]: \*Warning?

BytesWarning RuntimeWarning

DeprecationWarning SyntaxWarning

FutureWarning UnicodeWarning

ImportWarning UserWarning

PendingDeprecationWarning Warning

ResourceWarning

-new shortcuts in ipython terminal:(some of them work in vs code terminal as well)

Ctrl-r Reverse-search through command history

Ctrl-d Delete next character in line

Ctrl-k Cut text from cursor to end of line

Ctrl-u Cut text from beginning of line to cursor

Ctrl-y Yank (i.e. paste) text that was previously cut

Ctrl-t Transpose (i.e., switch) previous two characters

Ctrl-a Move cursor to the beginning of the line

Ctrl-e Move cursor to the end of the line

Ctrl-l Clear terminal screen

Ctrl-c Interrupt current Python command

Ctrl-d Exit IPython session

-%run <script.py>

will let the script run in ipython, and the methods are available to be called all along the session

-Another example of a useful magic function is %timeit, which will automatically determine the execution time of the single-line Python statement that follows it.1

-PASSING VALUES TO AND FROM THE SHELL

Shell Commands in IPython

Any command that works at the command-line can be used in IPython by prefixing it with the ! character. For example, the ls, pwd, and echo commands can be run as follows

In [1]: !ls

myproject.txt

In [2]: !pwd

/home/jake/projects/myproject

In [3]: !echo "printing from the shell"

printing from the shell

Besides %cd, other available shell-like magic functions are %cat, %cp, %env, %ls, %man, %mkdir, %more, %mv, %pwd, %rm, and %rmdir, any of which can be used without the % sign if automagic is on.

THIS ACCESS TO THE SHELL FROM WITHIN THE SAME TERMINAL WINDOW AS YOUR PYTHON SESSION MEANS THAT THERE IS A LOT LESS SWITCHING BACK AND FORTH BETWEEN INTERPRETER AND SHELL AS YOU WRITE YOUR PYTHON CODE.

==================================================================================================

1:numpy-

-NumPy: short for Numerical Python

provides an efficient interface to store and operate on dense data(many different pieces of the required information on a specific kind of a subject, no matter whatever the subject happens to be) buffers

NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

* Difference between numpy array and list

Diagram

Description automatically generated

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. **Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.**

* Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point) If we want to explicitly set the data type of the resulting array, we can use the dtype keyword
* np.array([1, 2, 3, 4], dtype='float32')
* nested lists(np.array([1,2,3])) result in multi-dimensional arrays
* np.full((3, 5), 3.14)

np.full((3, 5), 3.14)

array([[ 3.14, 3.14, 3.14, 3.14, 3.14],

[ 3.14, 3.14, 3.14, 3.14, 3.14],

[ 3.14, 3.14, 3.14, 3.14, 3.14]])

* - # Create a 3x3 array of normally distributed random values

# with mean 0 and standard deviation 1

np.random.normal(0, 1, (3, 3))

- NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

np.zeros(10, dtype='int16')

Or using the associated NumPy object:

np.zeros(10, dtype=np.int16)

-First let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* (we set the seed to an value so that random arrays generated will start from same number)with a set value in order to ensure that the same random arrays are generated each time this code is run:

In [1]:

**import** **numpy** **as** **np**

np.random.seed(0) *# seed for reproducibility*

x1 = np.random.randint(10, size=6)#(here the 10 indicates that numbers will be included from 0 to 10-1) *# One-dimensional array*

x2 = np.random.randint(10, size=(3, 4)) *# Two-dimensional array*

x3 = np.random.randint(10, size=(3, 4, 5)) *# Three-dimensional array*

-Each array has attributes ndim (the number of dimensions), shape (the size of each dimension), and size (the total size of the array), Another useful attribute is the dtype, the data type of the array. Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array.

-indexing in numpy is almost same as list indexing

-suppose if you have a 2 dimensional array, you can give x[1][1] or x[1,1] (this declaration is preferred)to access second element of second row

-Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that **if you attempt to insert a floating-point value to an integer array, the value will be silently truncated**. Don't be caught unaware by this behavior!

In [15]:

x1[0] = 3.14159 *# this will be truncated!*

X1

## -Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

x[start:stop:step]

If any of these are unspecified, they default to the values start=0, stop=*size of dimension*, step=1. We'll take

x = np.arange(10)

x

Out[16]:

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

x[1::2] *# every other element, starting at index 1*

Out[21]:

array([1, 3, 5, 7, 9])

**A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped**. This becomes a convenient way to reverse an array:

In [22]:

x[::-1] *# all elements, reversed*

Out[22]:

array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [23]:

x[5::-2] *# reversed every other from index 5*

Out[23]:

array([5, 3, 1])

-slicing a multi dimensional array

Arr[rowstart:rowend+1:rowstep,colstart:colend+1:colstep]

Ex1:arr[2:3:2,1:7:3]

Array output will be having row 2 and in row 2 further slicing happens which will include 1st,4th columns because we have colstep as 3

Ex2: ar=np.random.randint(10,size=(5,7))

Ar[::-1] will reverse rows

Ar[::,::-1] will reverse columns

Ar[::-1,::-1] will reverse rows and columns

Ex3: print(x2[:, 0])

Here step is not included for both row and col(when it is not included, by default it will be 1),the row does not have start and end, so all rows are included, but col has value 0, it means only first col is included, so first col values of all rows are returned

### -**Subarrays as no-copy views**

One important–and extremely useful–thing to know about array slices is that they return views rather than copies of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. If you take sub array and make some changes in it, the parent array will also be affected with the changes made.

### -**Creating copies of arrays**

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the copy() method:

In [35]:

x2\_sub\_copy = x2[:2, :2].copy()

print(x2\_sub\_copy)

[[99 5]

[ 7 6]]

If we now modify this subarray, the original array is not touched:

## -Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape method. For example, if you want to put the numbers 1 through 9 in a 3×33×3 grid, you can do the following:

In [38]:

grid = np.arange(1, 10).reshape((3, 3))

print(grid)

[[1 2 3]

[4 5 6]

[7 8 9]]

## -Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

### **Concatenation of arrays**

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines np.concatenate, np.vstack, and np.hstack. np.concatenate takes a tuple or list of arrays as its first argument, as we can see here:

In [43]:

x = np.array([1, 2, 3])

y = np.array([3, 2, 1])

np.concatenate([x, y])

Out[43]:

array([1, 2, 3, 3, 2, 1])

You can also concatenate more than two arrays at once:

In [44]:

z = [99, 99, 99]

print(np.concatenate([x, y, z]))

[ 1 2 3 3 2 1 99 99 99]

It can also be used for two-dimensional arrays:

In [45]:

grid = np.array([[1, 2, 3],

[4, 5, 6]])

In [46]:

*# concatenate along the first axis*

np.concatenate([grid, grid])

Out[46]:

array([[1, 2, 3],

[4, 5, 6],

[1, 2, 3],

[4, 5, 6]])

In [47]:

**# concatenate along the second axis (zero-indexed)**

np.concatenate([grid, grid], **axis**=1)

Out[47]:

array([[1, 2, 3, 1, 2, 3],

[4, 5, 6, 4, 5, 6]])

**For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack) and np.hstack (horizontal stack) functions:**

In [48]:

x = np.array([1, 2, 3])

grid = np.array([[9, 8, 7],

[6, 5, 4]])

*# vertically stack the arrays*

np.vstack([x, grid])

Out[48]:

array([[1, 2, 3],

[9, 8, 7],

[6, 5, 4]])

In [49]:

*# horizontally stack the arrays*

y = np.array([[99],

[99]])

np.hstack([grid, y])

Out[49]:

array([[ 9, 8, 7, 99],

[ 6, 5, 4, 99]])

Similary, np.dstack will stack arrays along the third axis.

-**skipped the split, dig deeper on split if you encounter work on it.**

**-**Computation on NumPy **arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations**, generally implemented through **NumPy's *universal functions* (ufuncs).** This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

Introducing UFuncs[¶](https://jakevdp.github.io/PythonDataScienceHandbook/02.03-computation-on-arrays-ufuncs.html#Introducing-UFuncs)

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a ***vectorized* operation**(allows the use of more optimal and pre-compiled functions and mathematical operations on NumPy array objects and data sequences, without using loops)**. This can be accomplished by simply performing an operation on the array, which will then be applied to each element**(notice the second print below, there values is a numpy array which is just being divided by 1.0 i.e, all elements of it are directly divided by 0 in dynamic mode like it happens in c, not in static mode like it happens in python). This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Compare the results of the following two:

In [3]:

print(compute\_reciprocals(values))

print(1.0 / values)

[ 0.16666667 1. 0.25 0.25 0.125 ]

[ 0.16666667 1. 0.25 0.25 0.125 ]

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

In [4]:

%**timeit** (1.0 / big\_array)

100 loops, best of 3: 4.6 ms per loop

Some examples:

* np.arange(5) / np.arange(1, 6)
* x = np.arange(9).reshape((3, 3))

2 \*\* x

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size. **Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.**

### -**Array arithmetic**

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

x = np.arange(4)

-(0.5\*x + 1) \*\* 2

Out[9]:

array([-1. , -2.25, -4. , -6.25])

Each of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the + operator is a wrapper for the add function:

i.e, x+2 is same as np.add(x,2)

The following table lists the arithmetic operators implemented in NumPy:

| **Operator** | **Equivalent ufunc** | **Description** |
| --- | --- | --- |
| + | np.add | Addition (e.g., 1 + 1 = 2) |
| - | np.subtract | Subtraction (e.g., 3 - 2 = 1) |
| - | np.negative | Unary negation (e.g., -2) |
| \* | np.multiply | Multiplication (e.g., 2 \* 3 = 6) |
| / | np.divide | Division (e.g., 3 / 2 = 1.5) |
| // | np.floor\_divide | Floor division (e.g., 3 // 2 = 1) |
| \*\* | np.power | Exponentiation (e.g., 2 \*\* 3 = 8) |
| % | np.mod | Modulus/remainder (e.g., 9 % 4 = 1) |

abs(arr) is as same as np.absolute(arr)

-trignometric and inverse trigonometric functions are also available in numpy

Ex: np.sin(theta), np.arcsin(theta)

-exponential functions in np:

x = [1, 2, 3]

print("x =", x)

print("e^x =", np.exp(x))

print("2^x =", np.exp2(x))

print("3^x =", np.power(3, x))

-log, base 2 log and base 10 log in numpy

x = [1, 2, 4, 10]

print("x =", x)

print("ln(x) =", np.log(x))

print("log2(x) =", np.log2(x))

print("log10(x) =", np.log10(x))

-reduce and accumulate are the aggregate functions that can be applied on add and multiply

Ex: np.multiply.reduce(x), np.add.accumulate(x)

There are equivalents to above methods:

np.sum, np.prod, np.cumsum, np.cumprod, np.min, np.max

-np.sum() is faster than sum() in python because numpy is written in c and c is a compiled language, compiled lngugage is faster most of the times

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

-By default, each NumPy aggregation function will return the aggregate over the entire array(even if its multi-d array)

<2d array>.sum()-> sum of all elements in 2d array

-Aggregation functions take an additional argument specifying the axis along which the aggregate is computed. For example, we can find the minimum value within each column by specifying axis=0

<2d array>.sum(axis=0)

**Axis explanation:**

**The way the axis is specified here can be confusing to users coming from other languages. The axis keyword specifies the dimension of the array that will be collapsed, rather than the dimension that will be returned. So specifying axis=0 means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated. axis=0 means row will be collapsed, columns will be considered for the operation(sum in the above example), vice a versa axis=1 means columns will be collapsed**

The above aggregate functions wont work, if there is an **nan** value in the array

| **Function Name** | **NaN-safe Version** | **Description** |
| --- | --- | --- |
| np.sum | np.nansum | Compute sum of elements |
| np.prod | np.nanprod | Compute product of elements |
| np.mean | np.nanmean | Compute mean of elements |
| np.std | np.nanstd | Compute standard deviation |
| np.var | np.nanvar | Compute variance |
| np.min | np.nanmin | Find minimum value |
| np.max | np.nanmax | Find maximum value |
| np.argmin | np.nanargmin | **Find index of minimum value** |
| np.argmax | np.nanargmax | **Find index of maximum value** |
| np.median | np.nanmedian | Compute median of elements |
| np.percentile | np.nanpercentile | Compute rank-based statistics of elements |
| np.any | N/A | Evaluate whether any elements are true |
| np.all | N/A | Evaluate whether all elements are true |

-broadcasting- suppose there is an numpy array and you want to add 5 to every eleent in the array, you can give <array>+5, this is broadcasting because, we are trying to apply addition of 5 just by using it once, it is similar to broadcasting message.

Suppose there is a 1d array and there is a 3d array,if you try to add them both, it will broadcast in such a way that,1d array is replicated 3 times to perform sum with 3 d array

Text

Description automatically generated

A picture containing text, keyboard, electronics

Description automatically generated

**The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.**

## Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

* Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
* Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
* Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

**Examples for rule 2 and 3 are below**

### **Broadcasting example 2**

Let's take a look at an example where both arrays need to be broadcast:

In [10]:

a = np.arange(3).reshape((3, 1))

b = np.arange(3)

Again, we'll start by writing out the shape of the arrays:

* a.shape = (3, 1)
* b.shape = (3,)

Rule 1 says we must pad the shape of b with ones:

* a.shape -> (3, 1)
* b.shape -> (1, 3)

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

* a.shape -> (3, 3)
* b.shape -> (3, 3)

Because the result matches, these shapes are compatible. We can see this here:

In [11]:

a + b

Out[11]:

array([[0, 1, 2],

[1, 2, 3],

[2, 3, 4]])

### **Broadcasting example 3**

Now let's take a look at an example in which the two arrays are not compatible:

In [12]:

M = np.ones((3, 2))

a = np.arange(3)

This is just a slightly different situation than in the first example: the matrix M is transposed. How does this affect the calculation? The shape of the arrays are

* M.shape = (3, 2)
* a.shape = (3,)

Again, rule 1 tells us that we must pad the shape of a with ones:

* M.shape -> (3, 2)
* a.shape -> (1, 3)

By rule 2, the first dimension of a is stretched to match that of M:

* M.shape -> (3, 2)
* a.shape -> (3, 3)

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

In [13]:

M + a

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-13-9e16e9f98da6> in <module>()

----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)

**-** Boolean masks are used to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

**Broadcasting can also be done using < and > operators:**

x = np.array([1, 2, 3, 4, 5])

**x<3**   
o/p:array([ True, True, False, False, False], dtype=bool)

the result is a **Boolean array**, and NumPy provides a number of straightforward patterns for working with these Boolean results.

similarly u can use >,>=,<=,==,!=

| **Operator** | **Equivalent ufunc** |  | **Operator** | **Equivalent ufunc** |
| --- | --- | --- | --- | --- |
| == | np.equal |  | != | np.not\_equal |
| < | np.less |  | <= | np.less\_equal |
| > | np.greater |  | >= | np.greater\_equal |

### **- Counting entries**

To count the number of True entries in a Boolean array, np.count\_nonzero is useful:

In [15]:

*# how many values less than 6?*

np.count\_nonzero(x < 6)

**here first it evaluates x<6 and bool array is generated, then it will count nonzero occurrences(i,e True),** **in this case, False is interpreted as 0, and True is interpreted as 1**

-The benefit of sum() is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

In [17]:

*# how many values less than 6 in each row?*

np.sum(x < 6, axis=1)

Out[17]:

array([4, 2, 2])

**-** If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any or np.all

Syntax: np.any(condition) or np.all(condition):

Np.any(X==6)

np.sum((inches > 0.5) & (inches < 1))

**-**

| **Operator** | **Equivalent ufunc** |  | **Operator** | **Equivalent ufunc** |
| --- | --- | --- | --- | --- |
| & | np.bitwise\_and |  | | | np.bitwise\_or |
| ^ | np.bitwise\_xor |  | ~ | np.bitwise\_not |

## Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our x array from before, suppose we want an array of all values in the array that are less than, say, 5:

**In [81]: x=np.array([[5, 0, 3, 3],**

**...: [7, 9, 3, 5],**

**...: [2, 4, 7, 6]])**

**In [82]: x**

**Out[82]:**

**array([[5, 0, 3, 3],**

**[7, 9, 3, 5],**

**[2, 4, 7, 6]])**

**In [83]: x<5**

**Out[83]:**

**array([[False, True, True, True],**

**[False, False, True, False],**

**[ True, True, False, False]])**

**In [84]: x[x<5]**

**Out[84]: array([0, 3, 3, 3, 2, 4])**

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True.

We are then free to operate on these values as we wish.

**Text

Description automatically generated**

**Notice the last example where we are trying to get the median of rain where we had the rain on all seasons except summer.**

**-when to use python operators “and” “or” and when to use “&” “|”:**

Refer to the detailed explanation as it might be important (

<https://jakevdp.github.io/PythonDataScienceHandbook/02.06-boolean-arrays-and-masks.html#Aside:-Using-the-Keywords-and/or-Versus-the-Operators-&/|>)

The difference is this: and and or gauge the truth or falsehood of entire object, while & and | refer to bits within each object.

When you use and or or, it's equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as True

Notice that the corresponding bits of the binary representation are compared in order to yield the result.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where 1 = True and 0 = False, and the result of & and | operates similarly to above:

**When operating with bits or bool data type if you get Value Error, then it might mean you are using either of the operator wrongly**

**-Fancy indexing:**

Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Ex:[x[3], x[7], x[2]]

Ex2: X = np.arange(12).reshape((3, 4))

row = np.array([0, 1, 2])

col = np.array([2, 1, 3])

X[row, col]

o/p: array([ 2, 5, 11])

ex:

Graphical user interface, application, Word

Description automatically generated

Ex:

Graphical user interface, text, application, email

Description automatically generated

Anomalies in NUmpy:

1:

Graphical user interface, text, application

Description automatically generated

2:

x = np.zeros(10)

In [89]: i = np.array([2, 1, 8, 4])

In [90]: i = [2, 3, 3, 4, 4, 4]

In [91]: x[i] += 1

In [92]: x

Out[92]: array([ 6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])

You might expect that x[3] would contain the value 2, and x[4] would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because x[i] += 1 is meant as a shorthand of x[i] = x[i] + 1. x[i] + 1 is evaluated, and then the result is assigned to the indices in x. With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the at() method of ufuncs (available since NumPy 1.8), and do the following:

Graphical user interface, text, application

Description automatically generated

The at() method does an in-place application of the given operator at the specified indices (here, i) with the specified value (here, 1). Another method that is similar in spirit is the reduceat() method of ufuncs, which you can read about in the NumPy documentation.

- By default np.sort uses an O[NlogN], quicksort algorithm, though mergesort and heapsort are also available. For most applications, the default quicksort is more than sufficient.

-sorting an numpy array is faster than sorting a list

-numpy methods which start with arg such as argsort and argmin will return index of the element

Text

Description automatically generated

- We can similarly create a **structured array** using a compound data type specification:

In [4]:

*# Use a compound data type for structured arrays*

data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),

'formats':('U10', 'i4', 'f8')})

*# Get names where age is under 30*

data[data['age'] < 30]['name']

- For clarity, numerical types can be specified using Python types or NumPy dtypes instead:

In [11]:

np.dtype({'names':('name', 'age', 'weight'),

'formats':((np.str\_, 10), int, np.float32)})

-np.array((“data type with data “or” any acceptable format”)) will be converted to numpy array.

- **numpy.argsort(*a*, *axis=- 1*, *kind=None*, *order=None*)**[**[source]**](https://github.com/numpy/numpy/blob/v1.22.0/numpy/core/fromnumeric.py#L1012-L1120)

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified by the *kind* keyword. It returns an array of indices of the same shape as *a* that index data along the given axis in sorted order.

-

==========================================================================================================

2:Pandas-

 Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame

Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

C and Cython are the sources on which Pandas is built

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices

Dataframe.sample(n) – will return some random n records from the dataframe

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

data = pd.Series([0.25, 0.5, 0.75, 1.0])

As we see in the output, the Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes.

Graphical user interface, text, application

Description automatically generated

The index is an array-like object of type pd.Index, which we'll discuss in more detail momentarily.

Graphical user interface, text, application, email

Description automatically generated

DataFrame.values attribute to return the numpy representation of the given DataFrame. In short words, it will remove the column names and returns a numpy array which resembles dataframe representation.

### **Series as generalized NumPy array**[**¶**](https://jakevdp.github.io/PythonDataScienceHandbook/03.01-introducing-pandas-objects.html#Series-as-generalized-NumPy-array)

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

Graphical user interface, text, application, email

Description automatically generated

data['b']

output: 0.50

### **Series as specialized dictionary**

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary:

Graphical user interface

Description automatically generated with low confidenceGraphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**data can be a scalar, which is repeated to fill the specified index**:

In [15]:

pd.Series(5, index=[100, 200, 300])

Out[15]:

100 5

200 5

300 5

dtype: int64

**data can be a dictionary, in which index defaults to the sorted dictionary keys(notice the keys are arranged as index):**

In [16]:

pd.Series({2:'a', 1:'b', 3:'c'})

Out[16]:

1 b

2 a

3 c

dtype: object

In each case, the index can be explicitly set if a different result is preferred:

pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])

Out[17]:

3 c

2 a

dtype: object

Notice that in this case, the Series is populated only with the explicitly identified keys.

the DataFrame can be thought of either **as a generalization of a NumPy array, or as a specialization of a Python dictionary**. We'll now take a look at each of these perspectives.

-Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

Similarly, we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier:

In [22]:

states['area']

Out[22]:

California 423967

Florida 170312

Illinois 149995

New York 141297

Texas 695662

Name: area, dtype: int64

-Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values:

In [25]:

pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])

a b c

0 1.0 2 NaN

1 NaN 3 4.0

#### **From a NumPy structured array**

We covered structured arrays in [Structured Data: NumPy's Structured Arrays](https://jakevdp.github.io/PythonDataScienceHandbook/02.09-structured-data-numpy.html). A Pandas DataFrame operates much like a structured array, and can be created directly from one:

In [28]:

A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])

A

Out[28]:

array([(0, 0.0), (0, 0.0), (0, 0.0)],

dtype=[('A', '<i8'), ('B', '<f8')])

In [29]:

pd.DataFrame(A)

## The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit index that lets you reference and modify data. This Index object is an interesting structure in itself, and it can be thought of either as an **immutable array** or as an ordered set (technically a multi-set, **as Index objects may contain repeated values**). Those views have some interesting consequences in the operations available on Index objects. As a simple example, let's construct an Index from a list of integers:

In [30]:

ind = pd.Index([2, 3, 5, 7, 11])

ind

Out[30]:

Int64Index([2, 3, 5, 7, 11], dtype='int64')

### **Index as immutable array**[**¶**](https://jakevdp.github.io/PythonDataScienceHandbook/03.01-introducing-pandas-objects.html#Index-as-immutable-array)

**The Index in many ways operates like an array**. For example, we can use standard Python indexing notation to retrieve values or slices:

In [31]:

ind[1]

Out[31]:

3

In [32]:

ind[::2]

Out[32]:

Int64Index([2, 5, 11], dtype='int64')

**Index objects also have many of the attributes familiar from NumPy arrays**:

In [33]:

print(ind.size, ind.shape, ind.ndim, ind.dtype)

5 (5,) 1 int64

**One difference between Index objects and NumPy arrays is that indices are immutable**–that is, they cannot be modified via the normal means:

In [34]:

ind[1] = 0

---------------------------------------------------------------------------

**TypeError** Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()

----> 1 ind[1] = 0

/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py in \_\_setitem\_\_(self, key, value)

**1243**

**1244** def \_\_setitem\_\_(self, key, value):

-> 1245 raise TypeError("Index does not support mutable operations")

**1246**

**1247** def \_\_getitem\_\_(self, key):

TypeError: Index does not support mutable operations

**This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.**

### **Index as ordered set**

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. **The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way**:

In [35]:

indA = pd.Index([1, 3, 5, 7, 9])

indB = pd.Index([2, 3, 5, 7, 11])

In [36]:

indA & indB *# intersection*

Out[36]:

Int64Index([3, 5, 7], dtype='int64')

In [37]:

indA | indB *# union*

Out[37]:

Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [38]:

indA ^ indB *# symmetric difference*

Out[38]:

Int64Index([1, 2, 9, 11], dtype='int64')

**These operations may also be accessed via object methods, for example indA.intersection(indB).**

- 'a' **in** data

If ‘a’ is in index of index object data ,then it will return True, else False

- the below operations on df are similar to dataframe:

data.keys()

Out[4]:

Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]:

list(data.items())

Out[5]:

[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]

Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value:

In [6]:

data['e'] = 1.25

data

Out[6]:

a 0.25

b 0.50

c 0.75

d 1.00

e 1.25

dtype: float64

### **Series as one-dimensional array**

**A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, slices, masking, and fancy indexing. Examples of these are as follows:**

**Notice the below examples:**

In [7]:

*# slicing by explicit index*

data['a':'c']

Out[7]:

a 0.25

b 0.50

c 0.75

dtype: float64

In [8]:

*# slicing by implicit integer index*

data[0:2]

Out[8]:

a 0.25

b 0.50

dtype: float64

In [9]:

*# masking*

data[(data > 0.3) & (data < 0.8)]

Out[9]:

b 0.50

c 0.75

dtype: float64

In [10]:

*# fancy indexing*

data[['a', 'e']]

Out[10]:

a 0.25

e 1.25

dtype: float64

**Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., data['a':'c']), the final index is included in the slice, while when slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice.**

### **Indexers: loc, iloc, and ix**

These slicing and indexing conventions can be a source of confusion. For example**, if your Series has an explicit integer index, an indexing operation such as data[1] will use the explicit indices, while a slicing operation like data[1:3] will use the implicit Python-style index.**

In [11]:

data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])

data

Out[11]:

1 a

3 b

5 c

dtype: object

In [12]:

***# explicit index when indexing***

data[1]

Out[12]:

'a'

In [13]:

***# implicit index when slicing***

data[1:3]

Out[13]:

3 b

5 c

dtype: object

**Because of this potential confusion in the case of integer indexes, Pandas provides some special indexer attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the Series.**

First, the loc attribute allows indexing and slicing that always references the **explicit index**:

In [14]:

data.loc[1]

Out[14]:

'a'

In [15]:

data.loc[3:5]

Out[15]:

Out[7]:

3 b

5 c

dtype: object

The iloc attribute allows indexing and slicing that always references the implicit Python-style index:

In [16]:

data.iloc[1]

Out[16]:

'b'

In [17]:

data.iloc[1:3]

Out[17]:

3 b

5 c

dtype: object

A third indexing attribute, ix, is a hybrid of the two, and for Series objects is equivalent to standard []-based indexing. The purpose of the ix indexer will become more apparent in the context of DataFrame objects, which we will discuss in a moment.

**One guiding principle of Python code is that "explicit is better than implicit."** The explicit nature of loc and iloc make them very useful in maintaining clean and readable code; **especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.**

### **DataFrame as a dictionary**

The first analogy we will consider is the DataFrame as a dictionary of related Series objects. Let's return to our example of areas and populations of states:

In [18]:

area = pd.Series({'California': 423967, 'Texas': 695662,

'New York': 141297, 'Florida': 170312,

'Illinois': 149995})

pop = pd.Series({'California': 38332521, 'Texas': 26448193,

'New York': 19651127, 'Florida': 19552860,

'Illinois': 12882135})

data = pd.DataFrame({'area':area, 'pop':pop})

data

Out[18]:

|  | **area** | **pop** |
| --- | --- | --- |
| **California** | 423967 | 38332521 |
| **Florida** | 170312 | 19552860 |
| **Illinois** | 149995 | 12882135 |
| **New York** | 141297 | 19651127 |
| **Texas** | 695662 | 26448193 |

**The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name:**

**In [19]:**

**data['area']**

Out[19]:

California 423967

Florida 170312

Illinois 149995

New York 141297

Texas 695662

Name: area, dtype: int64

**Equivalently, we can use attribute-style access with column names that are strings:**

**In [20]:**

**data.area**

Out[20]:

California 423967

Florida 170312

Illinois 149995

New York 141297

Texas 695662

Name: area, dtype: int64

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

In [21]:

data.area **is** data['area']

Out[21]:

True

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. For example, the DataFrame has a pop() method, so data.pop will point to this rather than the "pop" column:

In [22]:

data.pop **is** data['pop']

Out[22]:

False

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use data['pop'] = z rather than data.pop = z).

**Like with the Series objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:**

**In [23]:**

**data['density'] = data['pop'] / data['area']**

**data**

Out[23]:

|  | **area** | **pop** | **density** |
| --- | --- | --- | --- |
| **California** | 423967 | 38332521 | 90.413926 |
| **Florida** | 170312 | 19552860 | 114.806121 |
| **Illinois** | 149995 | 12882135 | 85.883763 |
| **New York** | 141297 | 19651127 | 139.076746 |
| **Texas** | 695662 | 26448193 | 38.018740 |

This shows a preview of the straightforward syntax of element-by-element arithmetic between Series objects; we'll dig into this further in [Operating on Data in Pandas](https://jakevdp.github.io/PythonDataScienceHandbook/03.03-operations-in-pandas.html).

Graphical user interface, text, application, email

Description automatically generated

-A.add(B, fill\_value=0)

Anove operation will replace Np.nan with 0

A picture containing table

Description automatically generated

-The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because it is a Python object, None cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects):

Not preferred because computation is not done in c, it is done inpython

Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

### **Duplicate indices**

One important difference between np.concatenate and pd.concat is that Pandas concatenation preserves indices, even if the result will have duplicate indices! Consider this simple example:

A picture containing table

Description automatically generated

Notice the repeated indices in the result. While this is valid within DataFrames, the outcome is often undesirable. pd.concat() gives us a few ways to handle it.

#### **Ignoring the index**

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the ignore\_index flag. With this set to true, the concatenation will create a new integer index for the resulting Series:

A picture containing text

Description automatically generated

Table

Description automatically generatedTable

Description automatically generated

For convenience, DataFrames implement the join() method, which performs a merge that defaults to joining on indices:

Table

Description automatically generated

Table

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Table

Description automatically generated



Text

Description automatically generated

Graphical user interface, table

Description automatically generated with medium confidence

The above code is really lengthy and a bit difficult

By using **Pivot tables** it can be done in a very easy manner.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

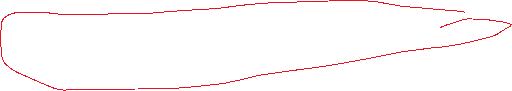
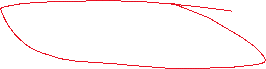
Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated



Graphical user interface, text

Description automatically generated

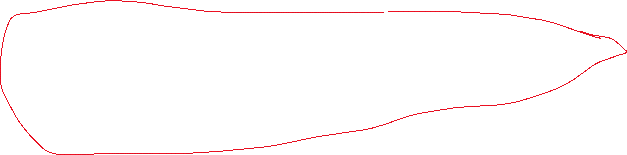


Text

Description automatically generated

Graphical user interface, text

Description automatically generated



***Get dummies:***

***Graphical user interface, text, application, email

Description automatically generated***

***Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application

Description automatically generated***

***Graphical user interface, text, application, email

Description automatically generated***

***Table

Description automatically generated with medium confidence***

Best Practices in Pandas:

A coding standard makes sure that all the developers working on the project are following certain specified guidelines. Formulating the code helps in easy understanding and maintenance of the code, and hence enhanced efficiency. It largely reduces the interdependence between modules and aids in better error handling.

Below are certain aspects of coding practices one should adhere to when developing scripts.

## 5.1. Merging Dataframes

Most of the data issues happen while merging datasets. Here are few basic checks.

1. **Determine key columns**
   * Look at the columns which are identically named using set(df1.columns).intersection(set(df2.columns))
   * On the columns that are identical, rename the columns which should not be a part of the key columns
   * Rename the key columns identically in both the tables
2. **Determine cardinality**
   * Identify the relationship between datasets. This can be better derived using business context and data dictionary provided by the business.

**See also**

Details on different types of relationships existing between datasets is provided [here](https://afteracademy.com/blog/what-are-the-different-types-of-relationships-in-dbms)

* + From the relationships between datasets, cardinality can be **1:m or 1:1 or m:1 or m:m**
  + For example, you are joining sales table with sku table on sku\_id. You can expect that sku table in universe of products and sales table is a subset. In this case the relationship is 1:m (with sku being left table)

1. **Determine the join type**
   * When there is key to key mismatches, use ta\_lib.core.utils.setanalyse()
   * When there is key combination mismatches, use ta\_lib.core.utils.setanalyse\_df()
   * Confirm & finalize with data owners
2. **Post Implementation Checks**
   * Record the number of columns before and after merging
   * Validate cardinality

## 5.2. Transformations

Following are the aspects that should be wary of when performing transformations.

1. Look for transformed column having values like inf or NA, log(0), div by zero, values outside limits etc.
2. **One hot encoding can fail in pipeline due to addition of new level or removal of existing level.**
   * Always ensure that the test and train data has the same set of levels. In case we have added or removed features in the test data, build a mechanism that drops/adds features that are in the test dataset.
3. Looping through dataframes/series is a strict NO
4. While binning numeric data, form *equi-width bins* following an increasing/decreasing distribution trend.
5. **Avoid using apply() with scalar functions. Try using series functions or write a custom function yourself**.
6. **Verify outputs**
   * Compare valid count of data points before and after transformation
   * While binning, group by bin and get min, max.
   * Check if the min and max values make sense. Say, for example, If you expect the transformed column to be positive, then min > 0

## 5.3. Intermediate data

It’s always a good idea to save cleaned tabular data using a storage format that supports the below aspects.

* Preserves the type information
* Language agnostic storage format
* Supports compression
* Supports customizing storage to optimize different data access patterns

For larger datasets, the last two points become crucial, Parquet is one such file format that is very popular for storing tabular data. Some of the nice features of Parquet are as follows:

* Similar to pickles & RDS datasets, but compatible with all languages
* Preserves the datatypes
* Compresses the data and reduces the filesize
* Good library support in Python and other languages
* As a columnar storage we can efficiently read fewer columns
* It also supports chunking data by groups of columns (for instance, by dates or a particular value of a key column) that makes loading subsets of the data fast.

**See also**

More details on [parquet](https://parquet.apache.org/documentation/latest/) can be found here.

## 5.4. Handling large files

* Use LTF viewer to open the file in a GUI
* Use [UNIX/Linux Tutorials](https://tigeranalytics-code-templates.readthedocs-hosted.com/en/latest/developer_guide/unix_tutorials.html#unix-tutorials) commands to fetch the file size, sample data etc.
* Load a sample of the dataset using [pandas.Dataframe.sample](https://tigeranalytics-code-templates.readthedocs-hosted.com/en/latest/developer_guide/%3Chttps:/pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sample.html%3E)
* Use dtypes=object for faster read of the data in pandas

**3:MatPlotLib:**

## plot Versus scatter: A Note on Efficiency

Aside from the different features available in plt.plot and plt.scatter, why might you choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, plt.plot can be noticeably more efficient than plt.scatter. The reason is that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In plt.plot, on the other hand, the points are always essentially clones of each other, so the work of determining the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

**Grid:**



grid is used to work with 3d plots , and you have only 2d data with you



why to choose seaborn over matplot:

An answer to these problems is [Seaborn](http://seaborn.pydata.org/). Seaborn provides an API on top of Matplotlib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality provided by Pandas DataFrames.

To be fair, the Matplotlib team is addressing this: it has recently added the plt.style tools discussed in [Customizing Matplotlib: Configurations and Style Sheets](https://jakevdp.github.io/PythonDataScienceHandbook/04.11-settings-and-stylesheets.html), and is starting to handle Pandas data more seamlessly. The 2.0 release of the library will include a new default stylesheet that will improve on the current status quo. But for all the reasons just discussed, Seaborn remains an extremely useful addon

the datasets and plot types available in Seaborn. Note that all of the following *could* be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood) but the Seaborn API is much more convenient.

* While binning numeric data, form *equi-width bins* following an increasing/decreasing distribution trend.



4:misc:

Context managers and managing resources:

Managing Resources : In any programming language, the usage of resources like file operations or database connections is very common. But these resources are limited in supply. Therefore, the main problem lies in making sure to release these resources after usage. If they are not released then it will lead to resource leakage and may cause the system to either slow down or crash. It would be very helpful if user have a mechanism for the automatic setup and teardown of resources.In Python, it can be achieved by the usage of **context managers** which facilitate the proper handling of resources. The most common way of performing file operations is by using the with keyword as shown below:

with open("test.txt") as f:

    data **=** f.read()

########################################################################

5:packaging: (https://packaging.python.org/en/latest/tutorials/packaging-projects/)

In packaging, while configuring metadata, there are 2 ways

There are two types of metadata: static and dynamic.

* Static metadata (setup.cfg): guaranteed to be the same every time. This is simpler, easier to read, and avoids many common errors, like encoding errors.
* Dynamic metadata (setup.py): possibly non-deterministic. Any items that are dynamic or determined at install-time, as well as extension modules or extensions to setuptools, need to go into setup.py.

Packages

A “package” is essentially a module, except it can have other modules (and indeed other packages) inside it.

A package usually corresponds to a directory with a file in it called \_\_init\_\_.py and any number of python files or other package directories:

a\_package

\_\_init\_\_.py

module\_a.py

a\_sub\_package

\_\_init\_\_.py

module\_b.py

- a package should definetly have setup.py so that it can have metadeta about the package and the metadata is as follows:

Version & package metadata

List of packages to include

List of other files to include

List of dependencies

List of extensions to be compiled

- to get all the dependencies or to export the environment to a yml file use(an env.yml file is needed because it is the list of packages that will be installed when you try to package your package):

conda env export > path of the file where you want to store dependencies (most probably an env.yml or environment.yml)

ex:

conda env export>env.yml

when you export all dependencies it will also copy the name of the environment in the env.yml file so that when you try to install dependencies in the yml file using the below command

**conda env create -f env.yml**

the name for above environment will be in the first line of env.yml

it will create a environment as mentioned in the env.yml and then install all the dependencies

in the same way if you want to generate the pip list use the below command:

pip list --format=freeze > requirements.txt

-pip show <package name> will give the information related to that particular package

Place where packages are installed (if you want to uninstall them or look the list of details related to package, you can refer this):

In conda env:

Syntax:

<Condaenv>\envs\env name\python version\sitepackages

Example:

miniconda3\envs\mle-dev\lib\python3.10\site-packages

-suppose you are having local imports and trying to import, and if you are getting module not found error, then you can use sys.path.append to add the path (path of the folder where the file you want to import exist)

When you are publishing a package to test.pypi.org(using the link- <https://packaging.python.org/en/latest/tutorials/packaging-projects/> ) if you want to include a particular set of files or folders or both explicitly(than the list of files that are already being packaged in to dist folder(which contains .gz and .whl files, .gz is the zipped format of whole project)) or to set the conditions similar to regex you can create manifest.in file and mention the conditions (as shown in the link <https://packaging.python.org/en/latest/guides/using-manifest-in/#using-manifest-in> )

recursive-include tests \*

include .gitignore (here we are adding all the files in test folder of cwd and .gitignore folder of cwd)

After running twine command you will get a whl file and a .tar.gz file, The tar.gz file is a [source archive](https://packaging.python.org/en/latest/glossary/#term-Source-Archive) whereas the .whl file is a [built distribution](https://packaging.python.org/en/latest/glossary/#term-Built-Distribution).

########################################################################

6:invoke:

Documentation: https://docs.pyinvoke.org/en/0.23.0/getting\_started.html

Video urls

* <https://www.youtube.com/watch?v=fqS2TBcxoeA>
* https://youtu.be/-BHverY7IwU?t=1120

Invoke is a package that is used to automate tasks, after installing it using **pip install invoke** , if you type **invoke <name of the task> <option1> <value1> <option2> <value2>…** (name of the task, options values are option) in cli, it will try to check for tasks.py file in the current directory and run it along with the parameters if passed any

Let’s say u opened a tasks.py file in cwd and typed the following content.

*from invoke import task*

*@task(name="webopener")*

*def open\_web\_page(self,url=None):*

*if url:*

*self.run(f"start {url}")*

*else:*

*print("no url passed, pass an url to open")*

if you have above syntax in tasks.py file (file should be in the cwd in which you will type **invoke** )and if you type **invoke webopener --url** [**www.google.com**](http://www.google.com)this will open an url in the browser

* You can have multiple tasks with or without name parameter(as discussed above),

Text

Description automatically generated

* If you type **invoke secondstep** in the cmd, it will execute firststep, then secondstep and then thirdstep because we have pre and post arguments with the task secondstep, we can also pass keyword arguments for the above example discussed in the video (<https://www.youtube.com/watch?v=fqS2TBcxoeA>)
* There are many other things that you can do with this, for example committing to a github repo(which is explained in the video)

# 7:Flask:

\_\_name\_\_ : special variable in python, that indicates the name of the module, \_\_name\_\_ is similar as \_\_main\_\_ , and behaves like a main if it is run with python directly, if the above module with \_\_name\_\_ is imported to somewhere else, then the name will be the name of the module. That’s why we have \_\_name\_\_ == \_\_main\_\_ so that if we import the module somewhere and we want to have some code which is to be executed by default, then you include that part of code under \_\_name\_\_ == \_\_main\_\_

@app.route(<path>) : we have various paths inside a website, ex: products webpage, contact us webpage and so on.. we handle them using the @app decorator

Decorators: a function in python that takes function as an argument and returns yet another function (another function as output), in simple words, it adds additional functionality to existing functions

Ex: @app.route(‘/’) – means the code written using this annotation will be for root page or home page of the website we are going to build

export FLASK\_APP= <flask python.py relative path> : this will set the application in the environment variables so that next time when you want to run the application, you can just type **flask run** in the console (to launch the app that we built), if you use the coding convention \_\_name\_\_==\_\_main\_\_, then include the below code and use **export FLASK\_DEBUG=1** in CLI, then by running python <scriptpath.py> you can run the application and the changes made will be reflected automatically because, you set flask\_debug and you gave the code app.run(debug=True)



after you make changes to your flask application, you need to rerun the application using **flask run** command, and then reload the webpage, to have changes in place

it maybe painful to run the code after changes by using **flask run** command every time, but you don’t have the need to do that if you run them in debug mode by setting environment variable **export FLASK\_DEBUG=1** now you don’t have the need to run the **flask run** command if you make changes, you can simply reload the page and check the changes you made

to perform **flask run** on a particular port use ex: flask run --host=0.0.0.0 --port=8085

In order to display HTML files using flask, you will need to use the **render\_template()** function, it only takes the name of the HTML file and the variable that you would like to pass to the template engine if any, By default flask looks for the template files in the templates directory, so now we need to create a templates directory in the same directory where you created the flask app

# 8: conda related:

* To update python in conda env: conda install python= <version you want to update>
* **env** command will list all environment variables in linux

# Misc:

* To find the location of installed package, use interpreter, then import <package>, then <package>.\_\_file\_\_
* When you give export FLASK\_APP=<script\_name.py> the script name is treated as package, when you try to give **flask run** to run the app, \_\_name\_\_ will store the name of the script , because the script is being treated as package, if you give python\_script.py **here, the script is being run as the main program** to run the app, then \_\_name\_\_ will store \_\_main\_\_ and then the code under \_\_name\_\_==’\_\_main\_\_’ will get executed
* Pip virtual environment
  + To create environment using pip:
    - python3 -m venv <env-name>
  + after you create environment it will create a folder **with the name of the environment you gave in python3 -m venv <env-name>** in the directory where you ran the command, and it will store all related info in the folder, so to activate the environment in the below command you will call **activate w**hich is stored in the folder that is created,
  + To activate pip environment in linux (should be done from the folder where it is created)
    - source <env-name>/bin/activate
  + else if you created the environment in the root folder and you are somewhere deep inside another folder, you can give
    - source ~/<environment-name>/bin/activate
  + to deactivate environment
    - type deactivate
* when you are facing package not found error even after installing them, then the interpreter is not able to find it, then you can check the python version that is set, and then install package in that version
  + syntax: <python version> -m pip install <package name>
* setting python 3.5 as default interpreter on centos:
  + sudo ln -fs /usr/bin/python3.5 /usr/bin/python
* when you activate both pip and conda virtual environment, afaik the environment which is activated later will be active until it is deactivated. You can check this by checking which python environment is active by giving “which python” command