FRAMEWORKS AND TECHNOLOGIES

OSCAR LLORENTE GONZALEZ

OLLORENTE@COMILLAS.EDU

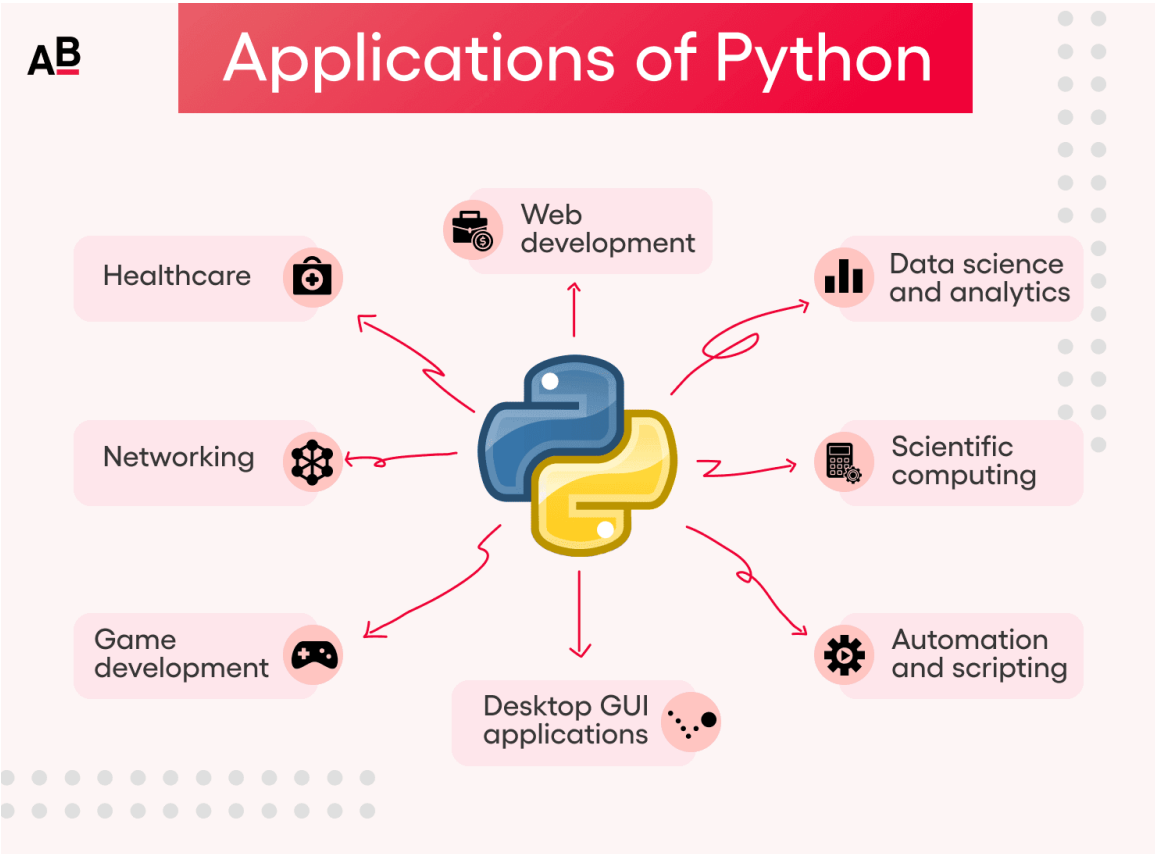
INDEX

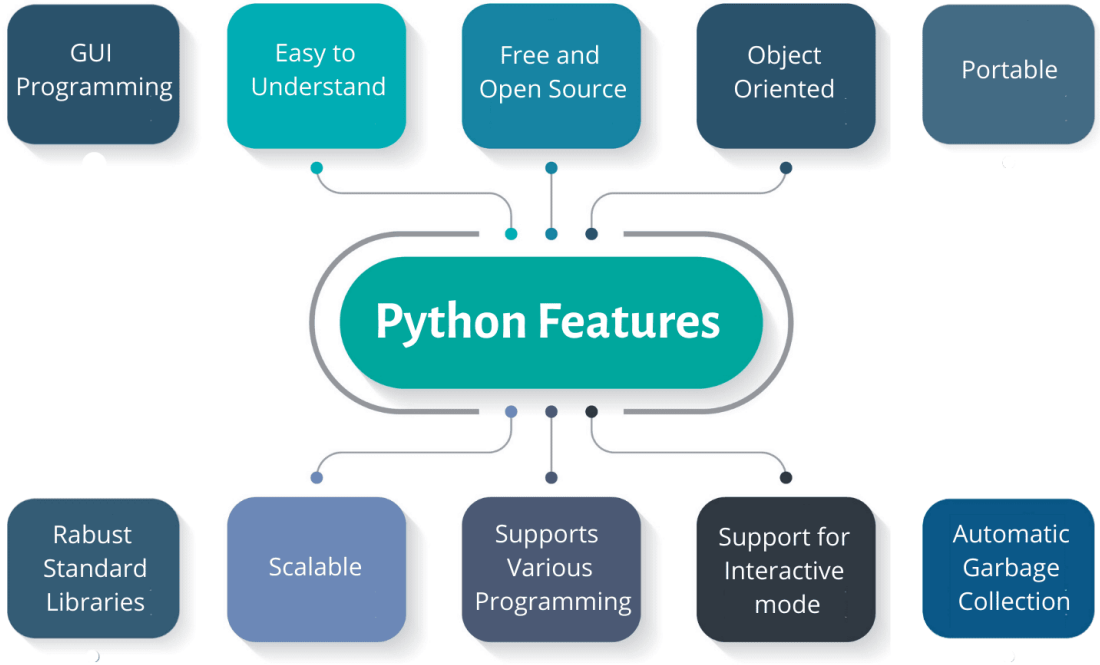
1 - PYTHON

PYTHON

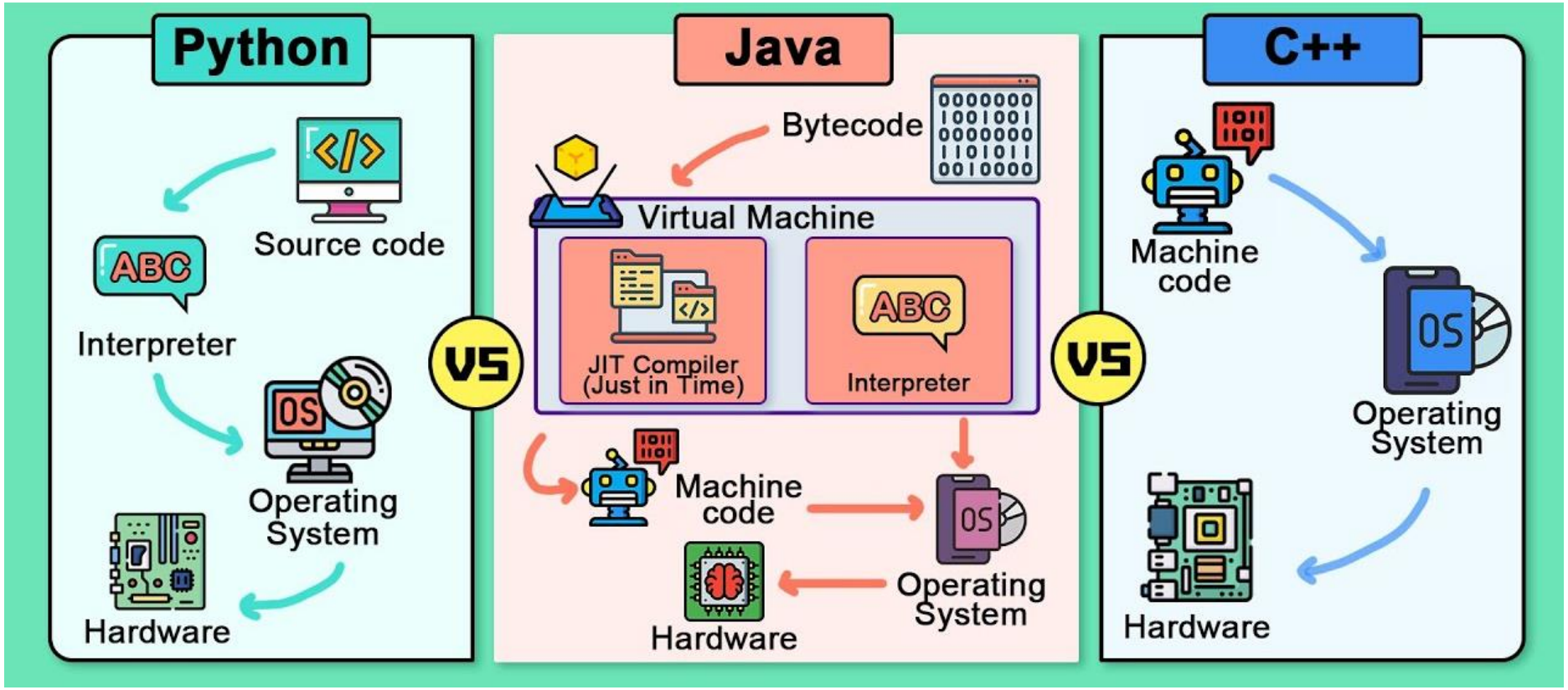
- How much do you know?
- Package's structure
- Type hints
- Black formatting
- PEP-8 standard
- Deep Learning Frameworks

WHAT IS PYTHON?

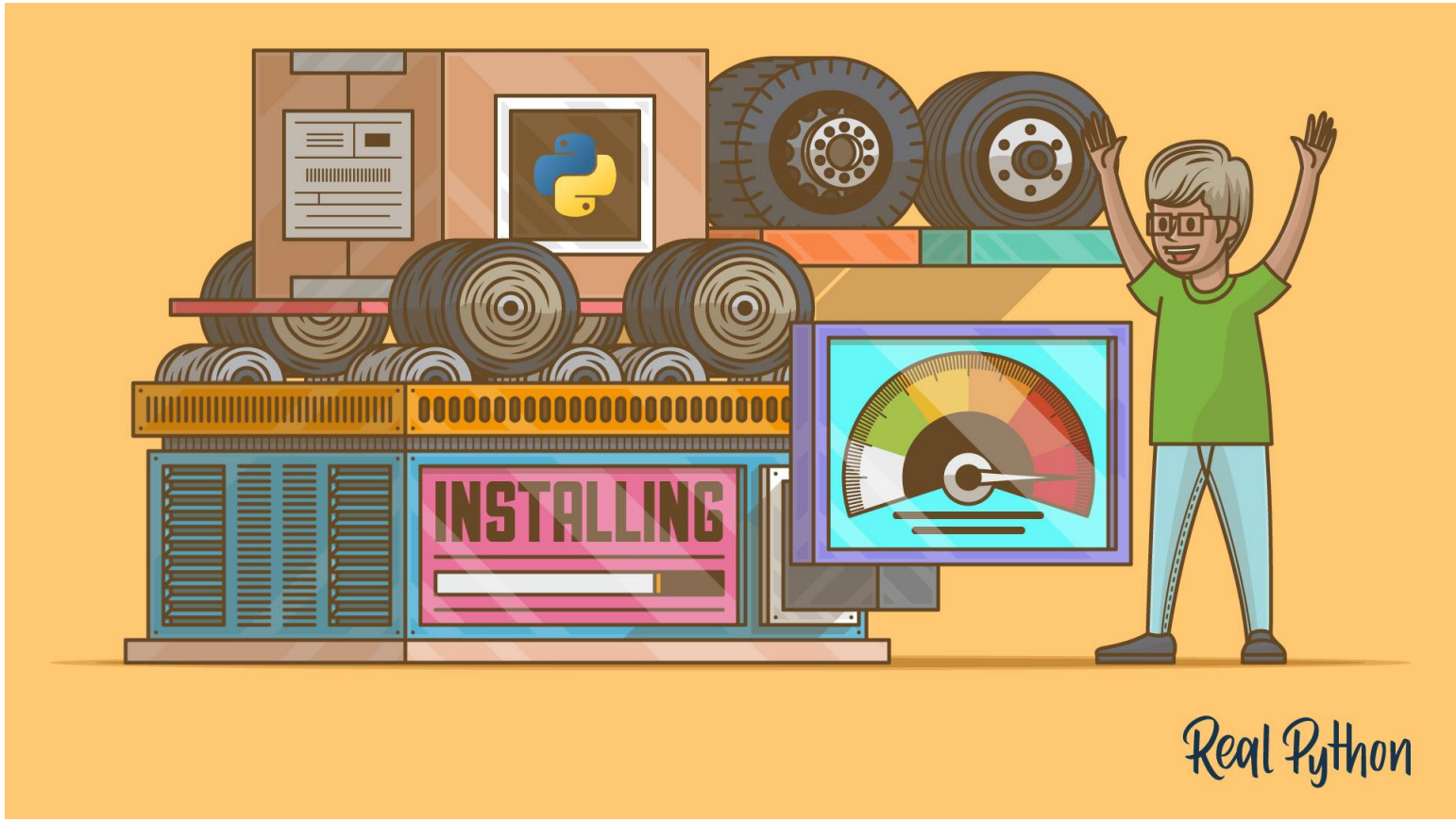




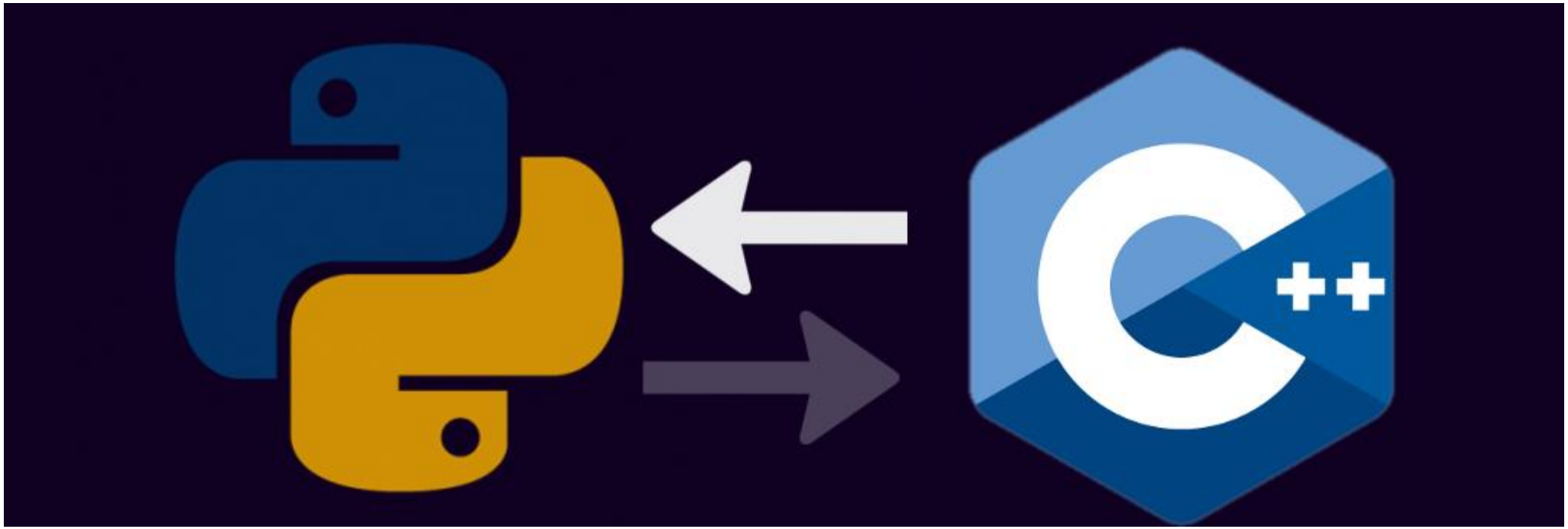
HOW DOES PYTHON WORK?



PYTHON LIBRARIES

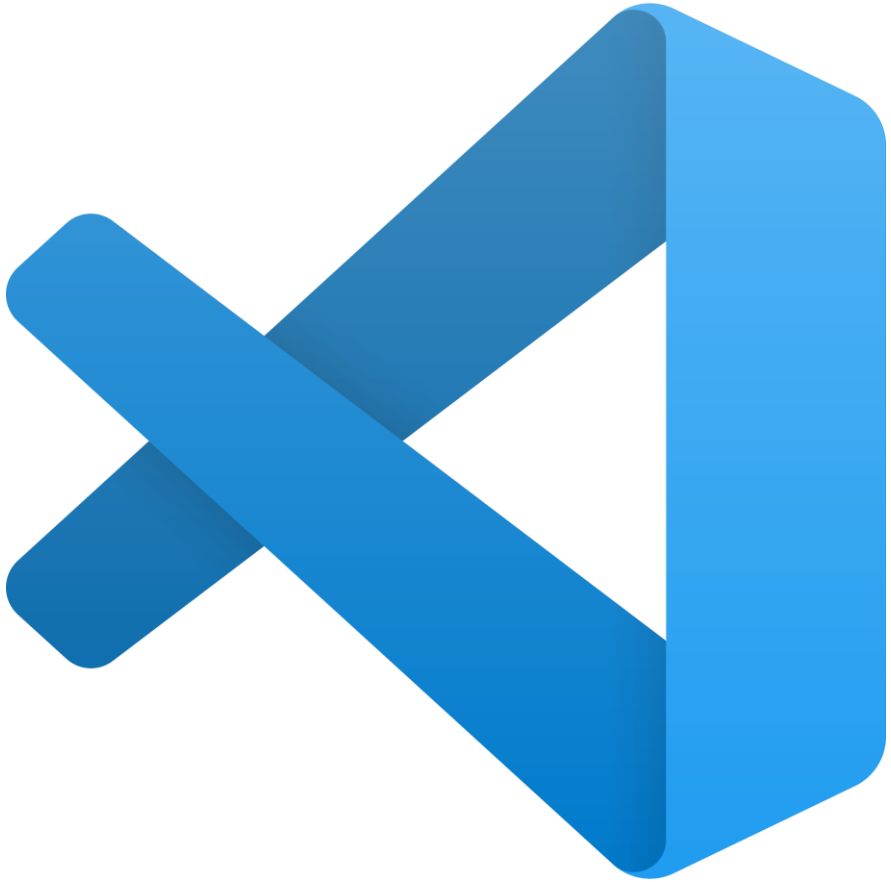


PYTHON AS THE INTERFACE



COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

VSCODE



NOTEBOOKS

PYTHON PACKAGES

```python
# absolute imports
from package.subpackage1.moduleY import *

# relative imports
from .moduleY import *
from ..subpackage2.moduleZ import *
```

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

DOCSTRINGS

```python
def main() -> None:
    """
    This function is the main program for training models

    Raises:
        ValueError: Invalid model name
    """
```

```python
def sum(var1: float, var2: float) -> float:
  """
  This method sums two variables

  Args:
      var1: first variable
      var2: second variable

  Returns
      sum of the two variables
  """

  # sum
  var3: float = var1 + var2

  return var3
```

```python
class MyClass:
    """
    This is the class to construct the model
    """

    # define variables
    var1: int
    var2: str

    def __init__(var1: int, var2: str) -> None:
      """
        This method is the constructor of the model

        Args:
            var1: first variable
            var2: second variable
      """

      # set attributes
      self.var1 = var1
      self.var2 = var2
```

TYPE HINTING

```python
from typing import Tuple, Optional, Literal, Final

var1: int = 1
var2: float = 0.5
var3: str = "hello"
var4: bool = True
var5: list[int] = [1, 2, 3]
var6: tuple[int, int] = (1, 2)
var7: tuple[int, ...] = (1, 2, 3, 4, 5, 6)
var8: Optional[str] = None
var9: str | None = None
var10: Union[int, float] = 4.5
var11: int | float = 4.5
var12: Literal["one",, "two", "three"] = "one"
var13: Final[int] = 2
```

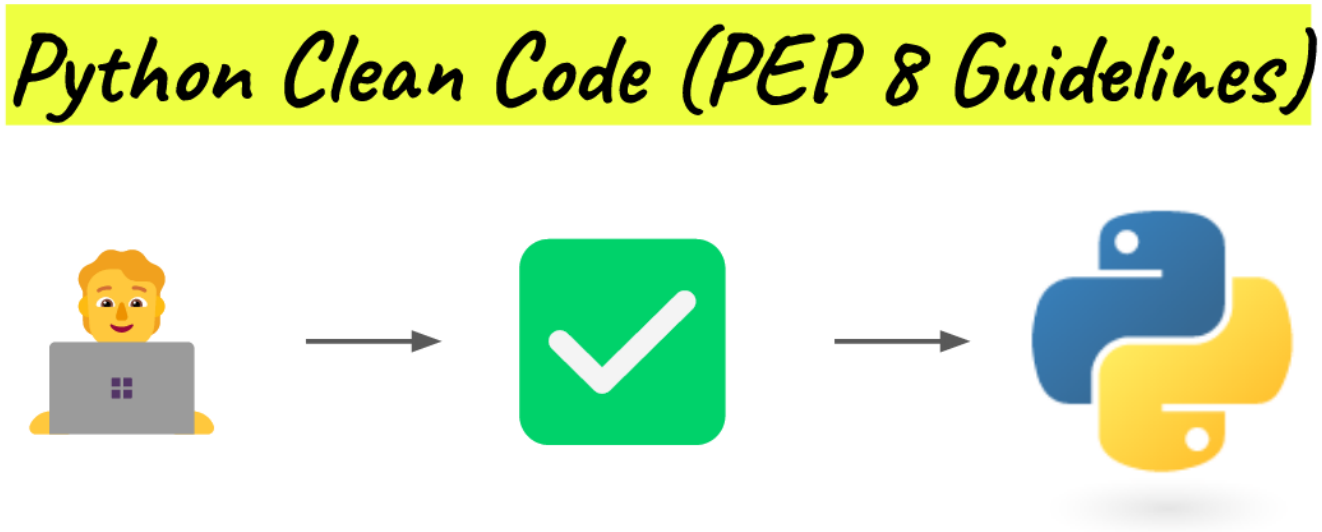
```python
def sum(var1: float, var2: float) -> float:
  """
  This method sums two variables

  Args:
      var1: first variable
      var2: second variable

  Returns
      sum of the two variables
  """

  # sum
  var3: float = var1 + var2

  return var3
```

```python
class MyClass:
    """
    This is the class to construct the model
    """

    # define variables
    var1: int
    var2: str

    def __init__(var1: int, var2: str) -> None:
      """
        This method is the constructor of the model

        Args:
            var1: first variable
            var2: second variable
      """

      # set attributes
      self.var1 = var1
      self.var2 = var2
```

MYPY


```
mypy --cache-dir=/dev/null .
```

PEP - 8



BLACK

```
black --line-length 79 .
```

pyproject.toml

```
[tool.black]
line-length = 99
```

- Line length of code - 79 (or until 99, decision of each team)
- Line length of docstrings - 72

INSTANCE METHODS

Python

```python
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'
```

CLASSMETHODS

Python

```python
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

STATIC METHODS

Python

```python
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

CALLABLE OBJECTS

Python

```python
# counter.py

class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1

    def __call__(self):
        self.increment()
```

Python

```
>>> callable(abs)
True
>>> callable(all)
True

>>> callable(greet)
True

>>> callable(SampleClass)
True

>>> callable(sample_instance)
False
```

ITERATORS

Python

```python
# sequence_iter.py

class SequenceIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            item = self._sequence[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

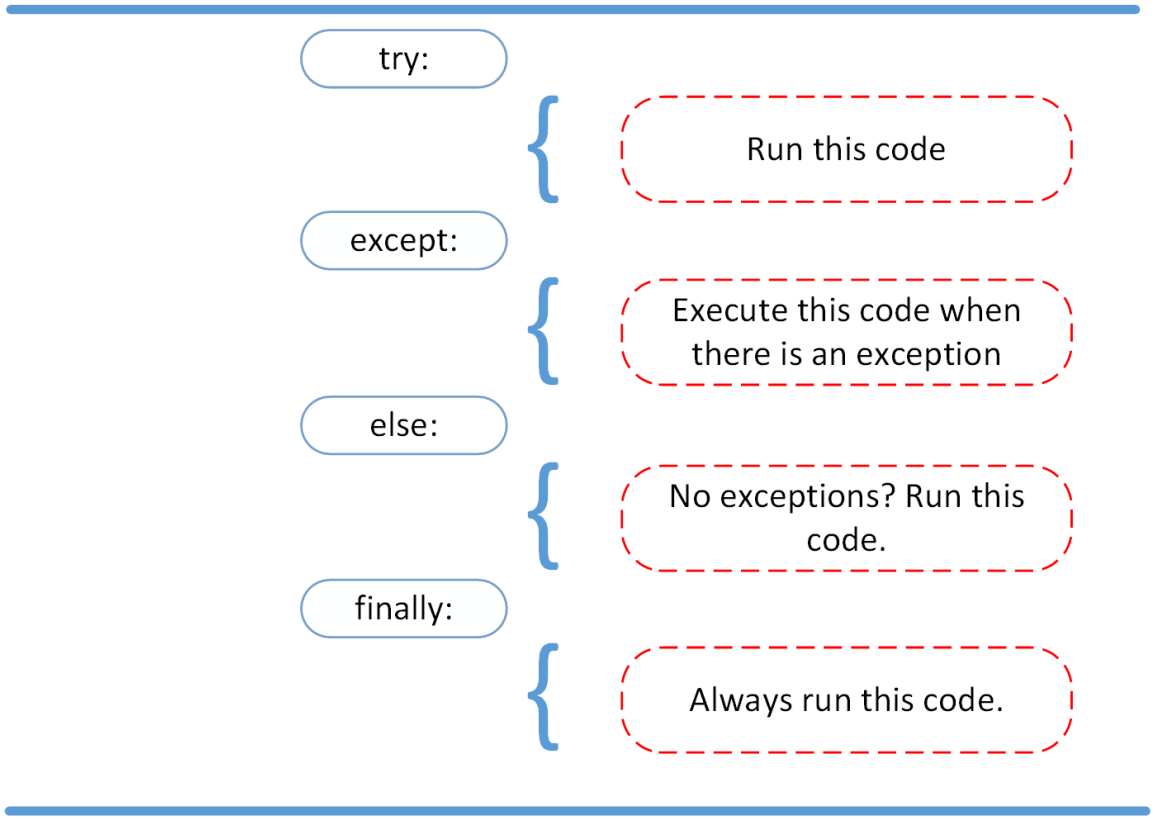
Method	Description
`.__iter__()`	Called to initialize the iterator. It must return an iterator object.
`.__next__()`	Called to iterate over the iterator. It must return the next value in the data stream.

GENERATORS

Python

```
>>> def sequence_generator(sequence):
...     for item in sequence:
...         yield item
...

>>> sequence_generator([1, 2, 3, 4])
<generator object sequence_generator at 0x108cb6260>

>>> for number in sequence_generator([1, 2, 3, 4]):
...     print(number)
...
1
2
3
4
```

EXCEPTIONS


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

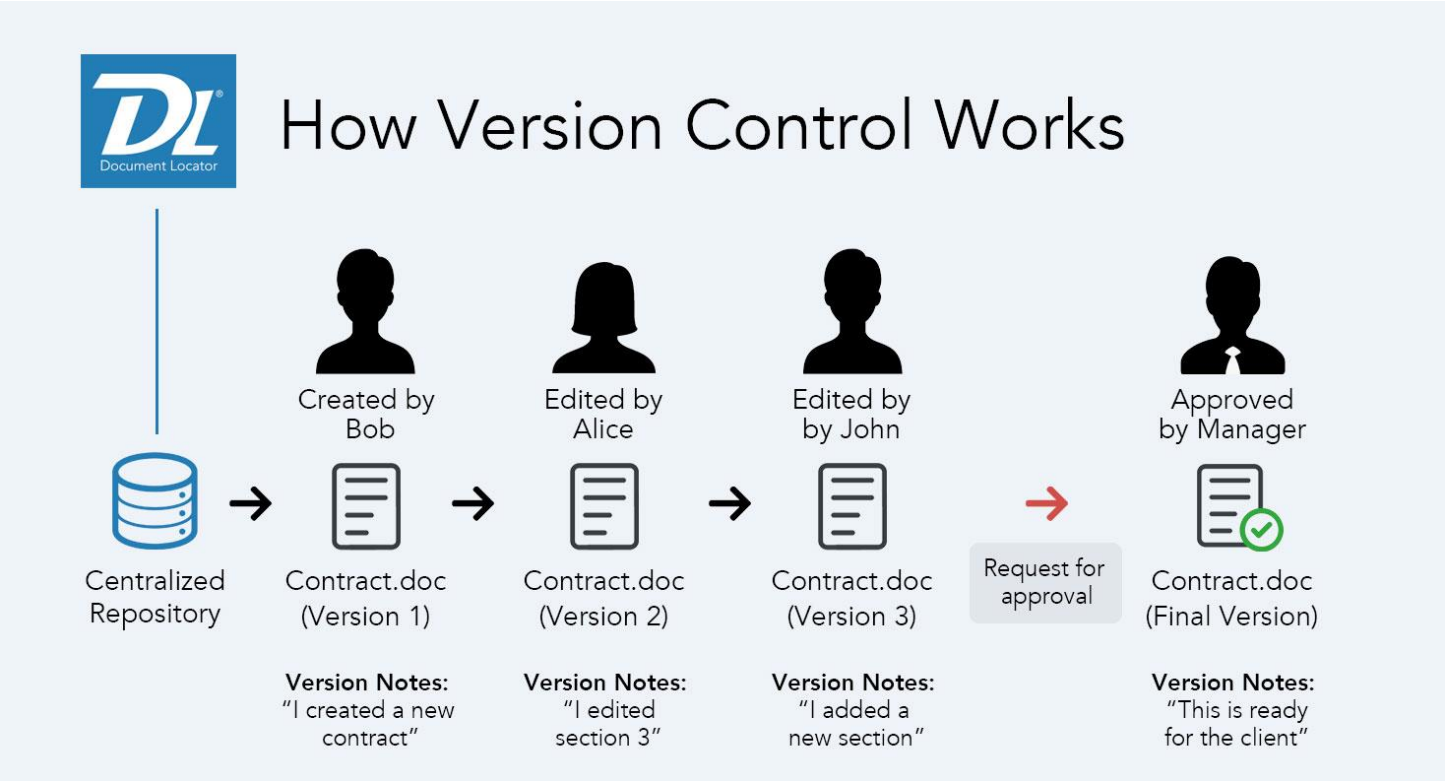
DECORATORS

Python hello_decorator.py

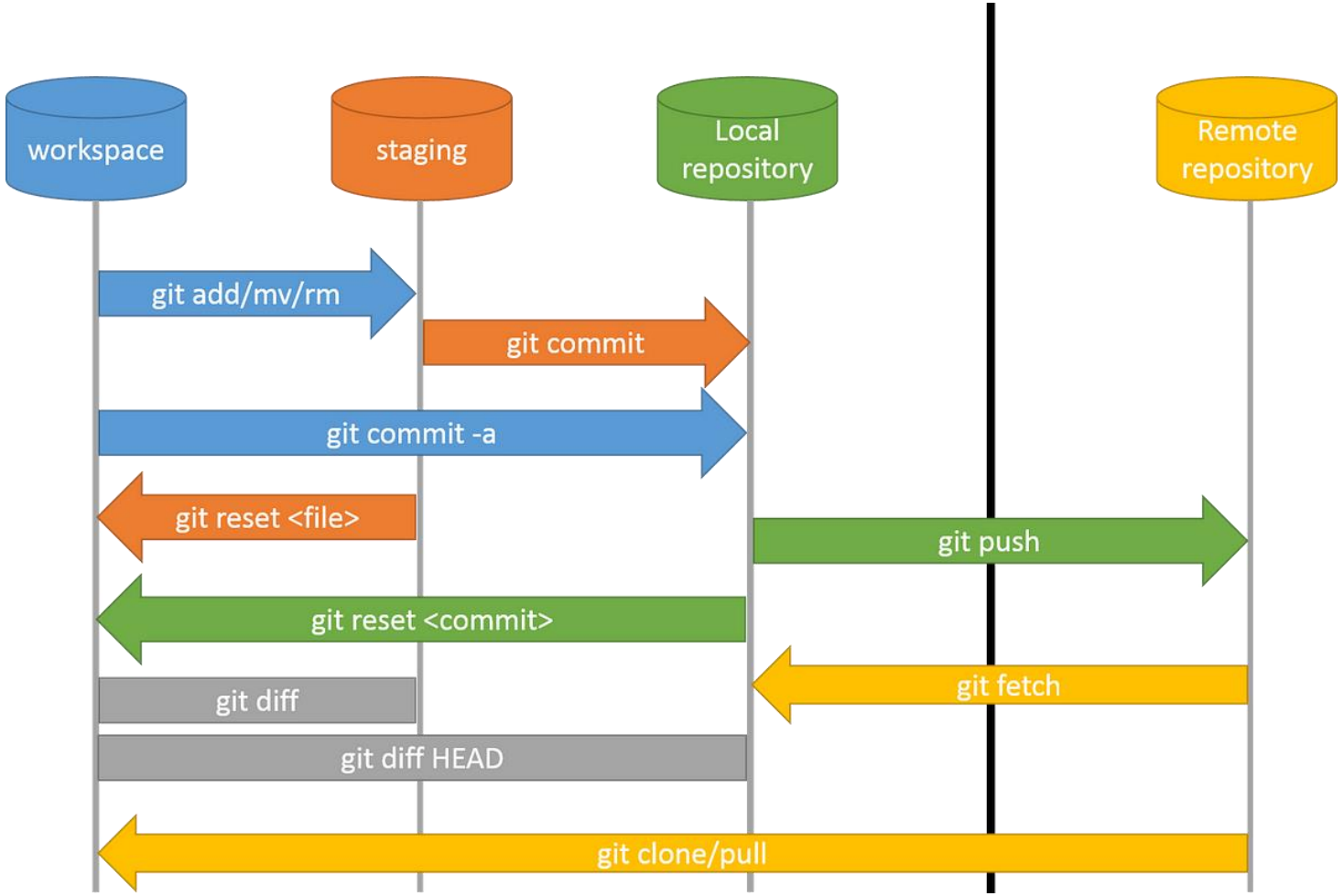
```python
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = decorator(say_whee)
```

2 - GIT

VERSION CONTROL



GIT STRUCTURE

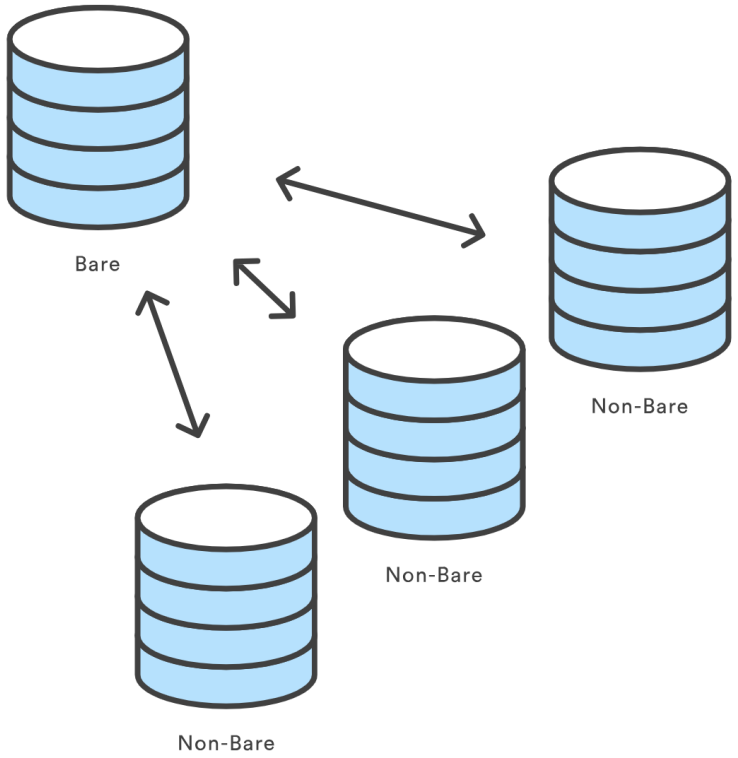


GIT INIT

```
git init
```

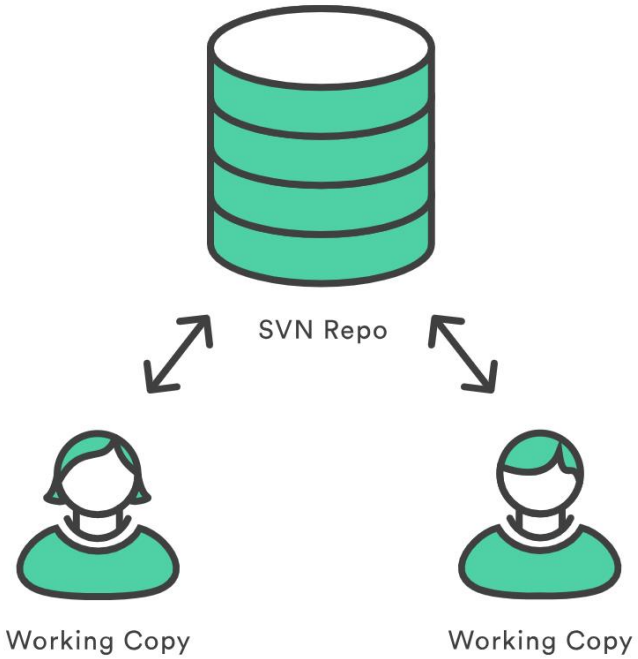
```
git init <directory>
```

```
git init --bare <directory>
```

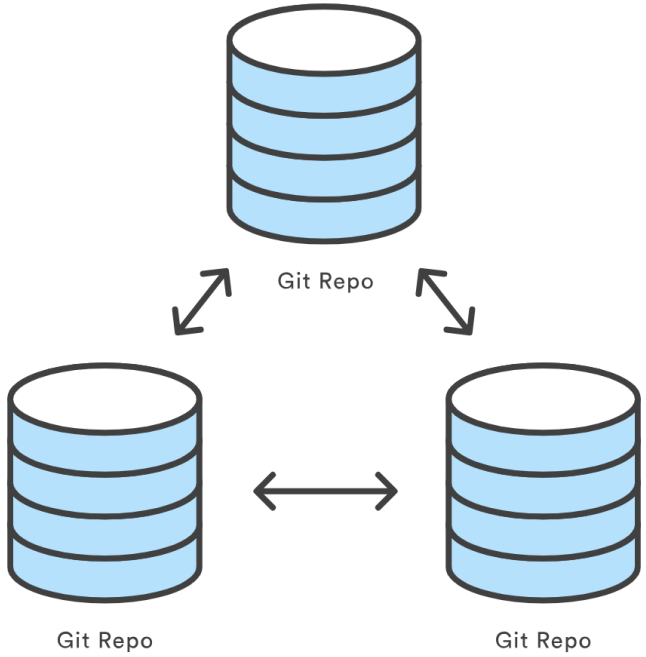


GIT CLONE

Central-Repo-to-Working-Copy Collaboration



Repo-To-Repo Collaboration



GIT CLONE

Cloning to a specific folder

```
git clone <repo> <directory>
```

Clone the repository located at `<repo>` into the folder called `~<directory>!` on the local machine.

Cloning a specific tag

```
git clone --branch <tag> <repo>
```

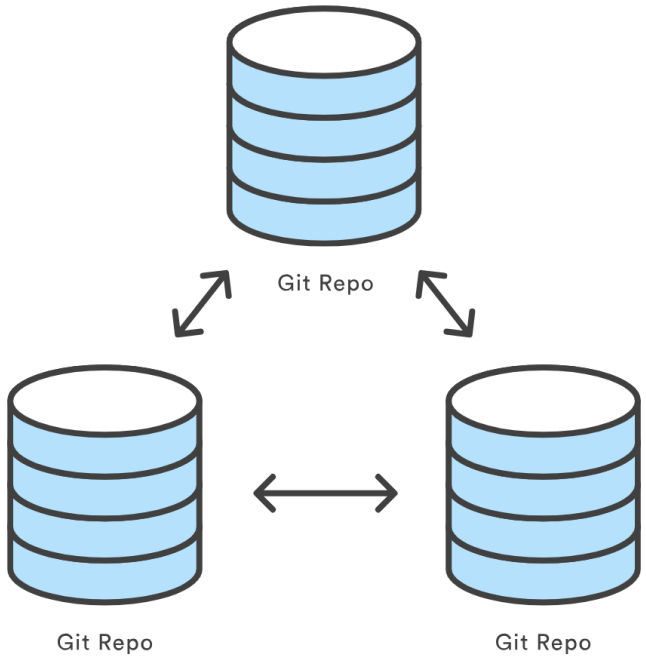
Clone the repository located at `<repo>` and only clone the ref for `<tag>`.

Shallow clone

```
git clone -depth=1 <repo>
```

Clone the repository located at `<repo>` and only clone the history of commits specified by the option depth=1. In this example a clone of `<repo>` is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

Repo-To-Repo Collaboration



GIT CONFIG

Writing a value

Expanding on what we already know about `git config`, let's look at an example in which we write a value:

```
git config --global user.email "your_email@example.com"
```

This example writes the value `your_email@example.com` to the configuration name `user.email`. It uses the `--global` flag so this value is set for the current operating system user.

git config levels and files

Before we further discuss `git config` usage, let's take a moment to cover configuration levels. The `git config` command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

- `--local`

By default, `git config` will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository `git config` gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: `.git/config`

- `--global`

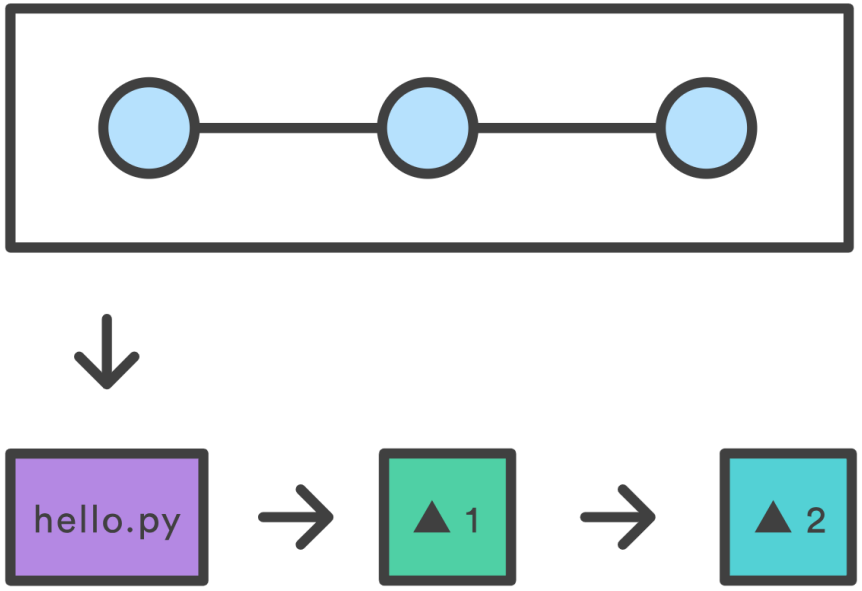
Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory. `~ /.gitconfig` on unix systems and `C:\Users\\.gitconfig` on windows

- `--system`

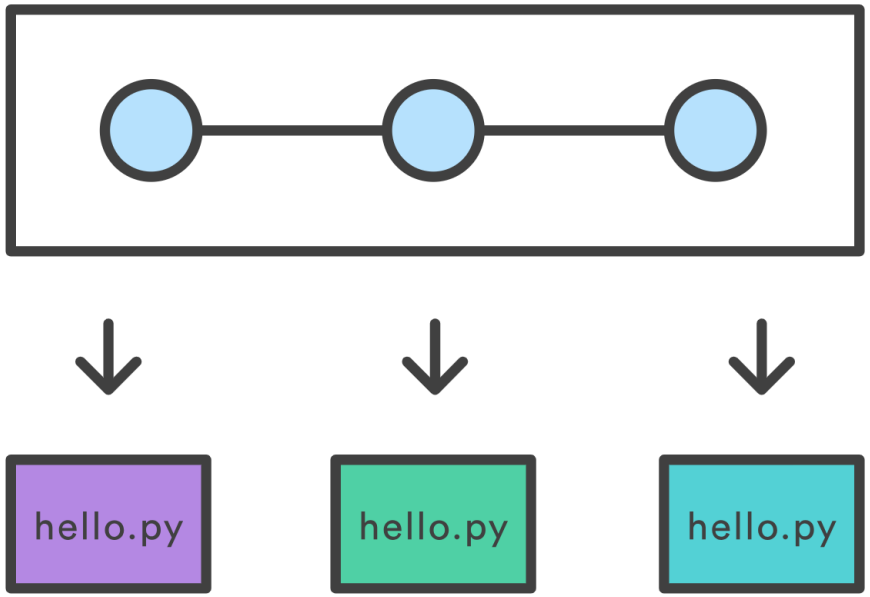
System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. The system level configuration file lives in a `gitconfig` file off the system root path. `$(prefix)/etc/gitconfig` on unix systems. On windows this file can be found at `C:\Documents and Settings\All Users\Application Data\Git\config` on Windows XP, and in `C:\ProgramData\Git\config` on Windows Vista and newer.

GIT COMMIT

Recording File Diffs (SVN)



Recording Snapshots (Git)



GIT COMMIT

```
git commit
```

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you've entered a message, save the file and close the editor to create the actual commit.

```
git commit -a
```

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with `git add` at some point in their history).

```
git commit -m "commit message"
```

A shortcut command that immediately creates a commit with a passed commit message. By default, `git commit` will open up the locally configured text editor, and prompt for a commit message to be entered. Passing the `-m` option will forgo the text editor prompt in-favor of an inline message.

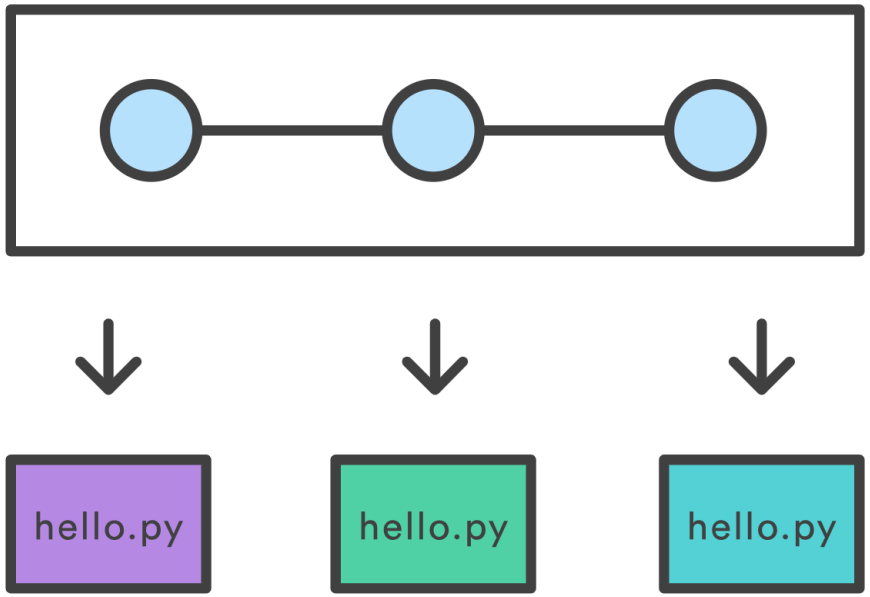
```
git commit -am "commit message"
```

A power user shortcut command that combines the `-a` and `-m` options. This combination immediately creates a commit of all the staged changes and takes an inline commit message.

```
git commit --amend
```

This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit. This command will open up the system's configured text editor and prompt to change the previously specified commit message.

Recording Snapshots (Git)



GITIGNORE

Pattern	Example matches	Explanation*

`**/logs`	`logs/debug.log`
	`logs/monday/foo.bar`
	`build/logs/debug.log`
		You can prepend a pattern with a double asterisk to match directories anywhere in the repository.

`**/logs/debug.log`	`logs/debug.log`
	`build/logs/debug.log`
	but not
	`logs/build/debug.log`
		You can also use a double asterisk to match files based on their name and the name of their parent directory.

`*.log`	`debug.log`
	`foo.log`
	`.log`
	`logs/debug.log`
		An asterisk is a wildcard that matches zero or more characters.

`*.log`
`!important.log`
	`debug.log`
	but not
	`logs/debug.log`
		Prepending an exclamation mark to a pattern negates it. If a file matches a pattern, but *also* matches a negating pattern defined later in the file, it will not be ignored.

`/debug.log`	`debug.log`
	but not
	`logs/debug.log`
		Patterns defined after a negating pattern will re-ignore any previously negated files.

`debug.log`	`debug.log` `logs/debug.log`	Prepending a slash matches files only in the repository root.

`debug?.log`	`debug0.log`
	`debugg.log`
	but not
	`debug10.log`
		A question mark matches exactly one character.

Pattern	Example matches	Explanation*

`logs`	`logs`
	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs`
	`build/logs/debug.log`
		If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named *logs* are ignored

`logs/`	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs/foo.bar`
	`build/logs/latest/debug.log`
		Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and subdirectories – will be ignored

`logs/` `!logs/important.log`	`logs/debug.log` `logs/important.log`	Wait a minute! Shouldn't `logs/important.log` be negated in the example on the left Nope! Due to a performance-related quirk in Git, you can not negate a file that is ignored due to a pattern matching a directory

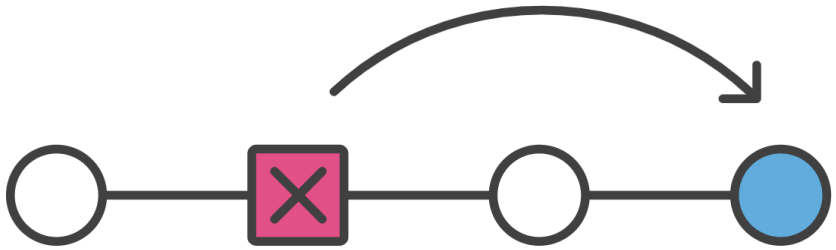
`logs/**/debug.log`	`logs/debug.log`
	`logs/monday/debug.log`
	`logs/monday/pm/debug.log`
		A double asterisk matches zero or more directories.

`logs/*day/debug.log`	`logs/monday/debug.log`
	`logs/tuesday/debug.log`
	but not
	`logs/latest/debug.log`
		Wildcards can be used in directory names as well.

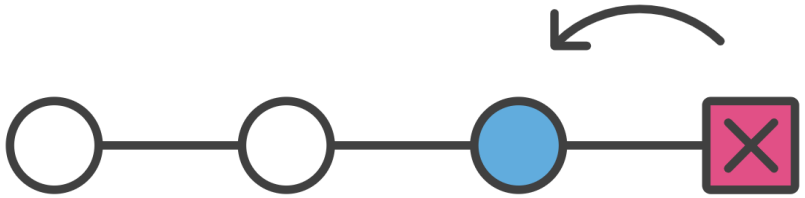
`logs/debug.log`	`logs/debug.log`
	but not
	`debug.log`
	`build/logs/debug.log`
		Patterns specifying a file in a particular directory are relative to the repository root. (You can prepend a slash if you like, but it doesn't do anything special.)

GIT REVERT VS RESET

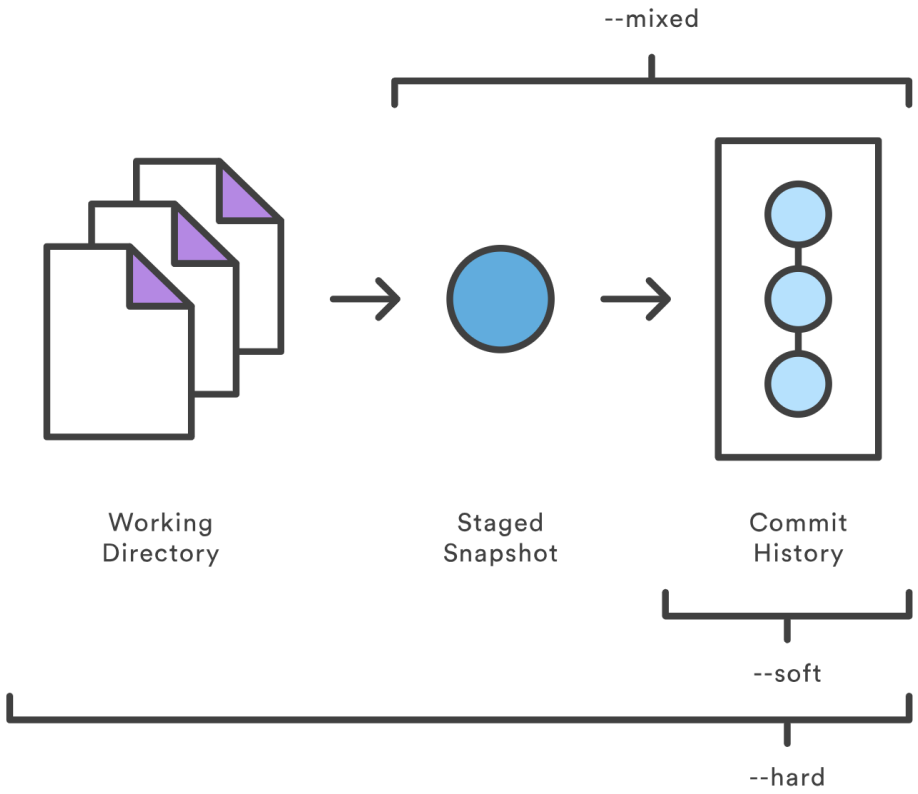
Reverting



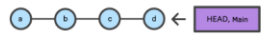
Resetting



GIT RESET

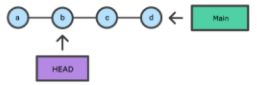


At a surface level, `git reset` is similar in behavior to `git checkout`. Where `git checkout` solely operates on the `HEAD` ref pointer, `git reset` will move the `HEAD` ref pointer and the current branch ref pointer. To better demonstrate this behavior consider the following example:



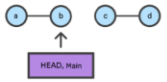
This example demonstrates a sequence of commits on the `main` branch. The `HEAD` ref and `main` branch ref currently point to commit d. Now let us execute and compare, both `git checkout b` and `git reset b`.

git checkout b



With `git checkout`, the `main` ref is still pointing to `d`. The `HEAD` ref has been moved, and now points at commit `b`. The repo is now in a 'detached `HEAD`' state.

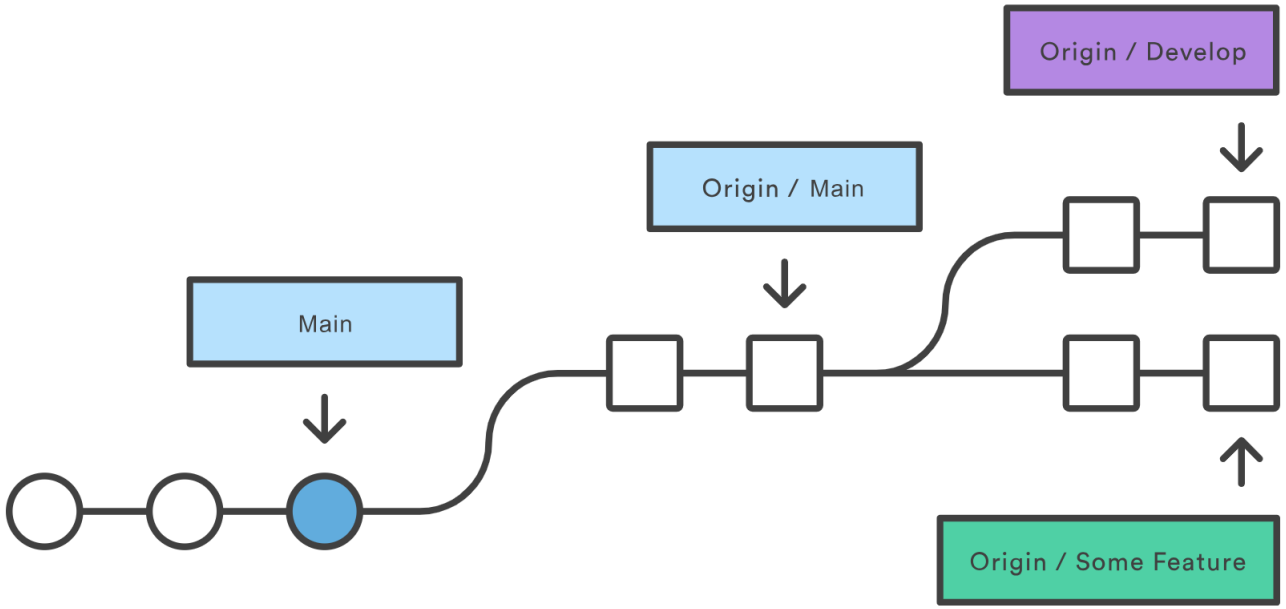
git reset b



Comparatively, `git reset`, moves both the `HEAD` and branch refs to the specified commit.

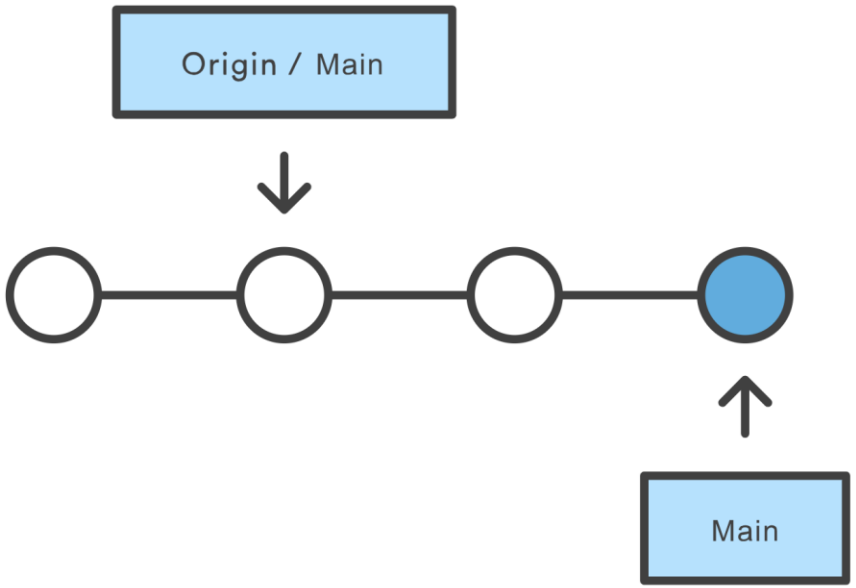
In addition to updating the commit ref pointers, `git reset` will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments `--soft`, `--mixed`, and `--hard` direct how to modify the Staging Index, and Working Directory trees.

GIT FETCH

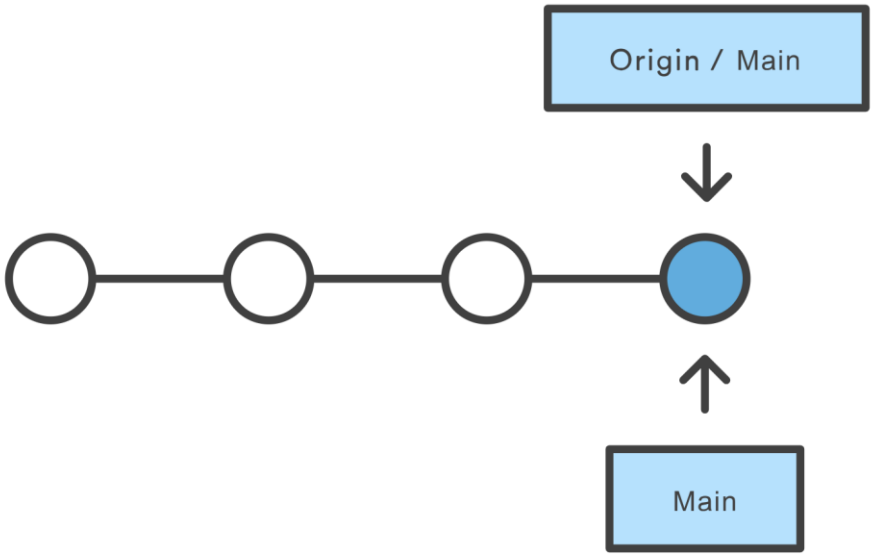


GIT PUSH

Before Pushing



After Pushing



GIT PUSH

```
git push <remote> <branch>
```

Push the specified branch to , along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.

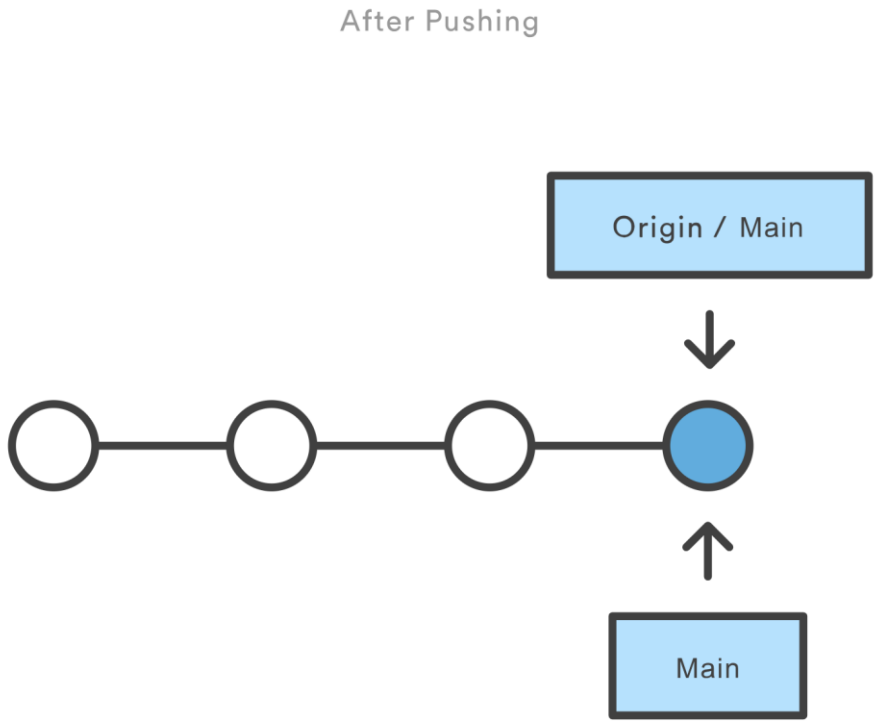
```
git push <remote> --force
```

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the `--force` flag unless you're absolutely sure you know what you're doing.

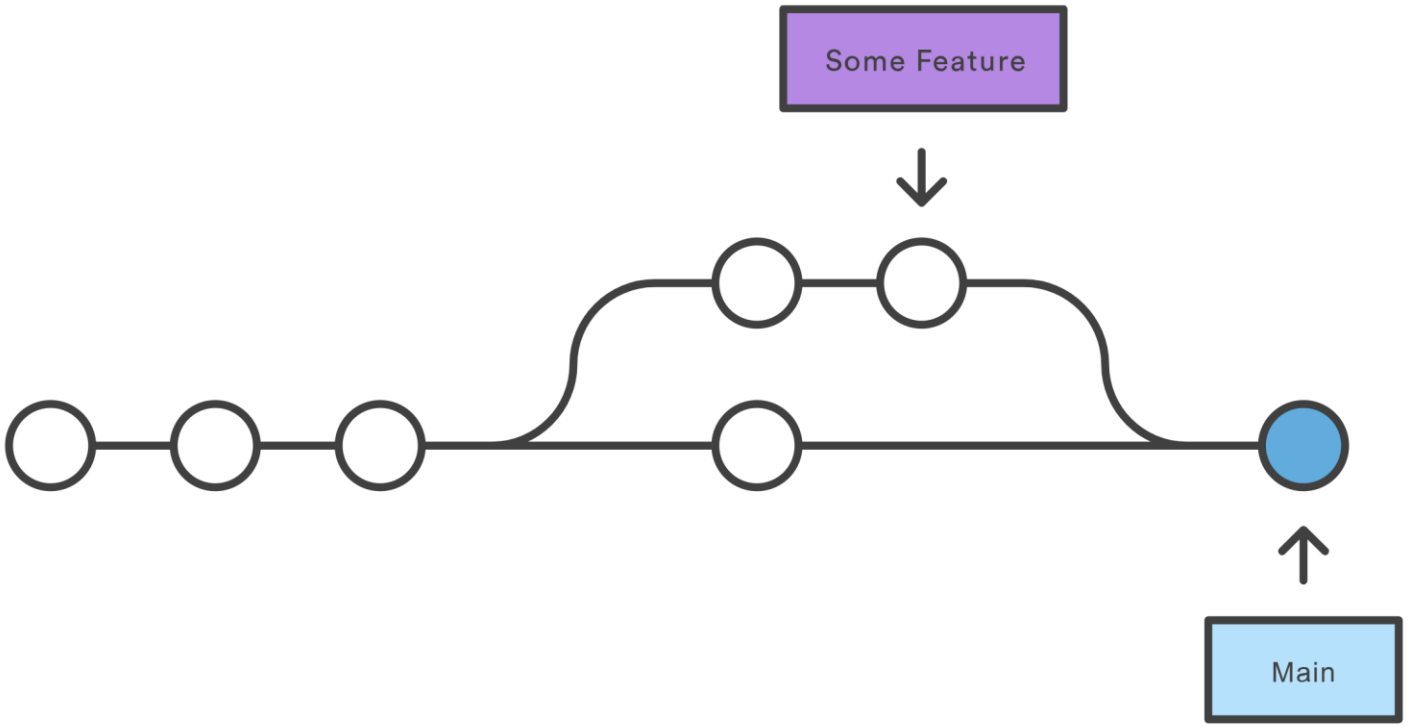
Push all of your local branches to the specified remote.

```
git push <remote> --tags
```

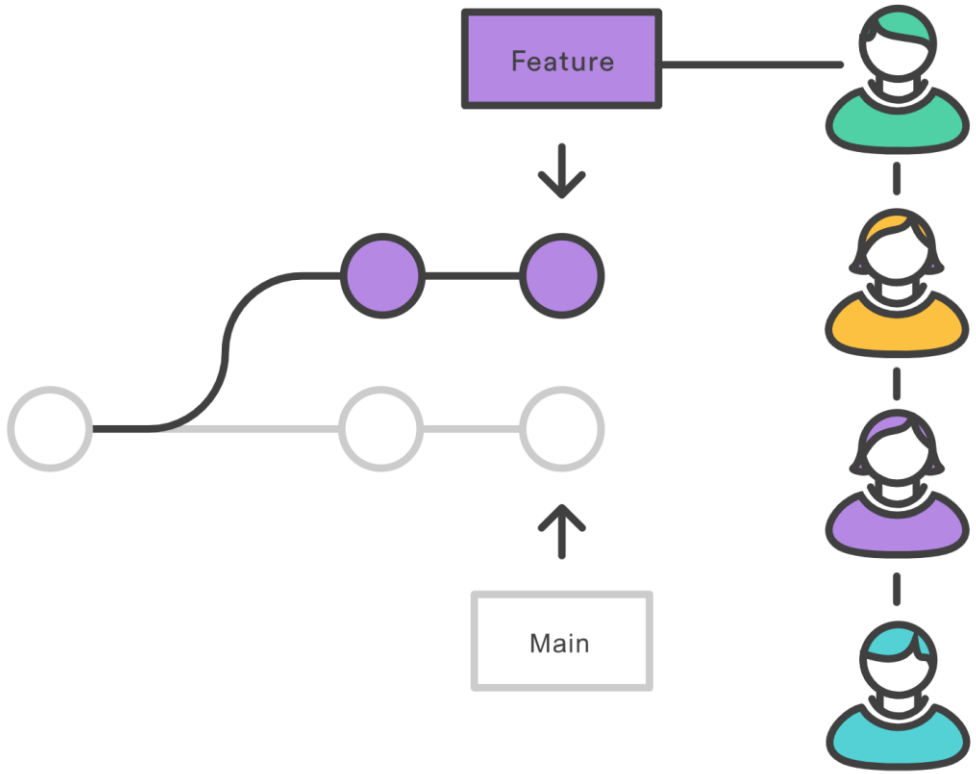
Tags are not automatically pushed when you push a branch or use the `--all` option. The `--tags` flag sends all of your local tags to the remote repository.



GIT MERGE



MERGE/PULL REQUESTS



3 - ARRAYS

WHAT ARE ARRAYS?

Image

3-dimensional: $\mathbf{I} \in \mathbb{R}^{H \times W \times C}$

Video

4-dimensional: $\mathbf{V} \in \mathbb{R}^{T \times H \times W \times C}$

HOW TO CREATE ARRAYS

```
data = np.array([1,2])
```

data
1
2

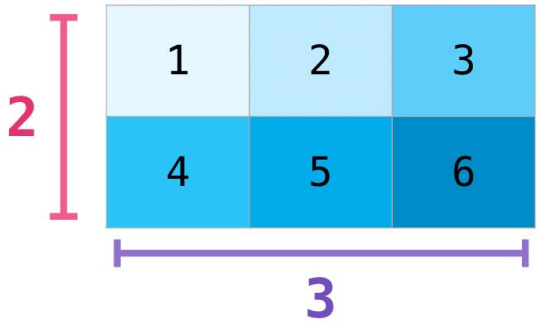
```
ones = np.ones(2)
```

ones
1
1

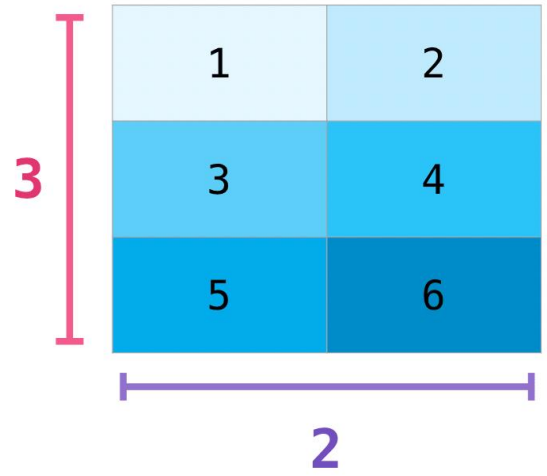
data

1
2
3
4
5
6

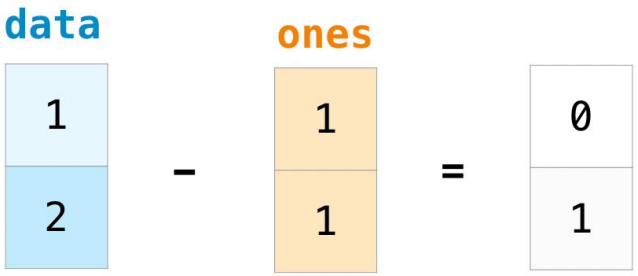
data.reshape(2,3)

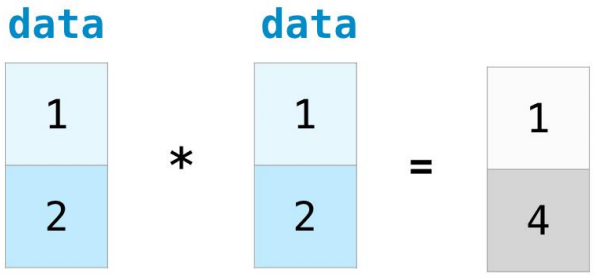


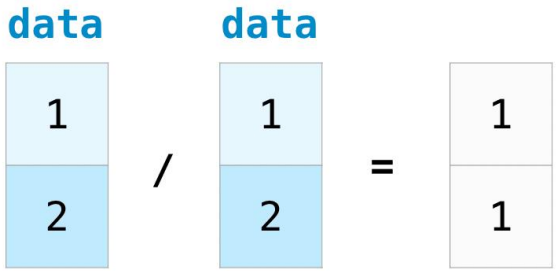
data.reshape(3,2)



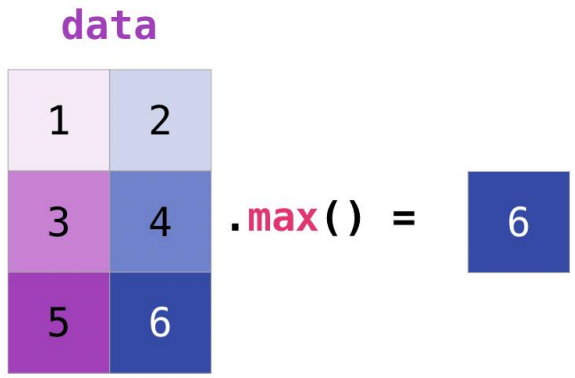
ARRAY OPERATIONS

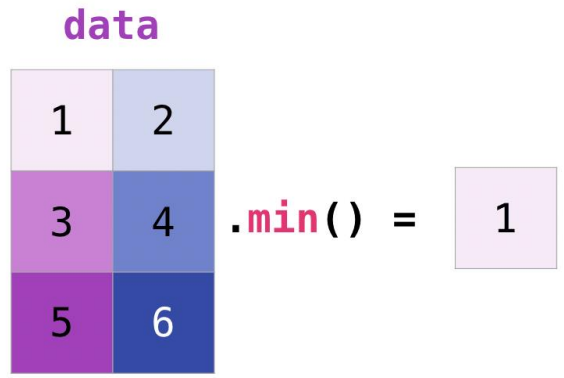


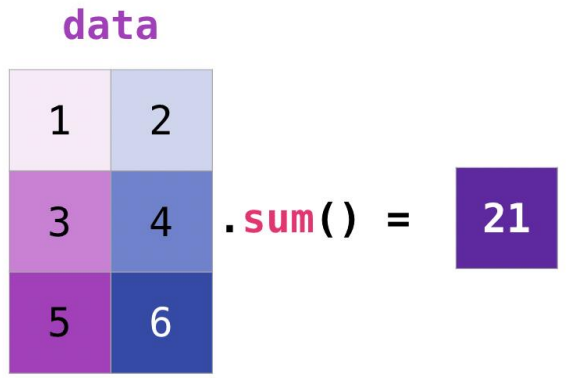




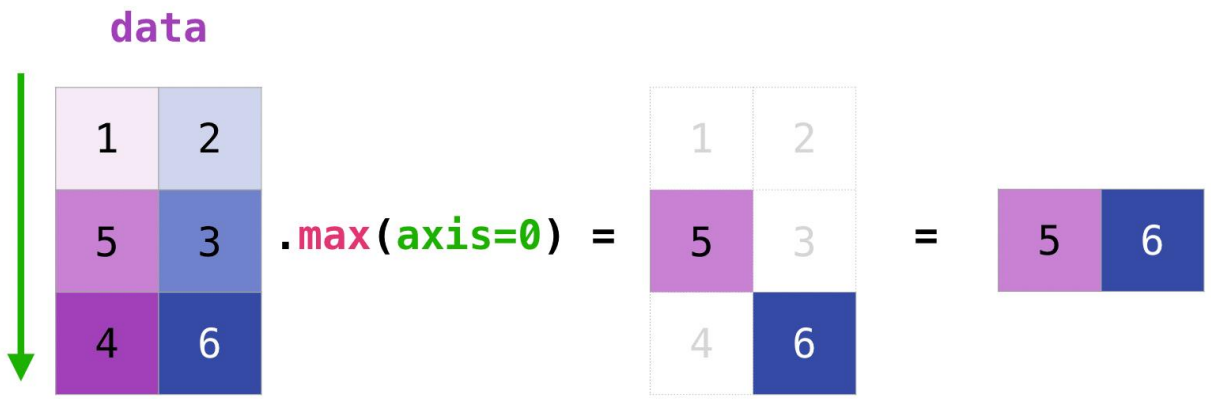
ARRAY OPERATIONS

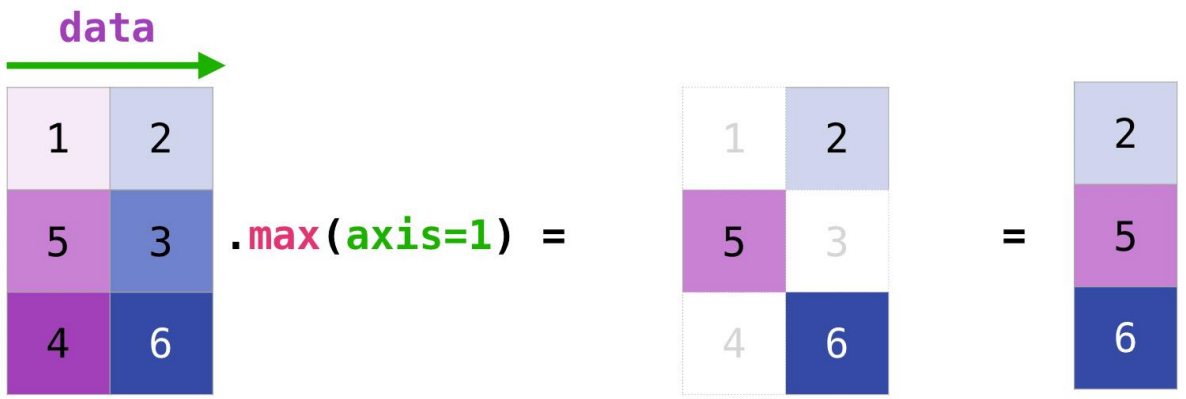




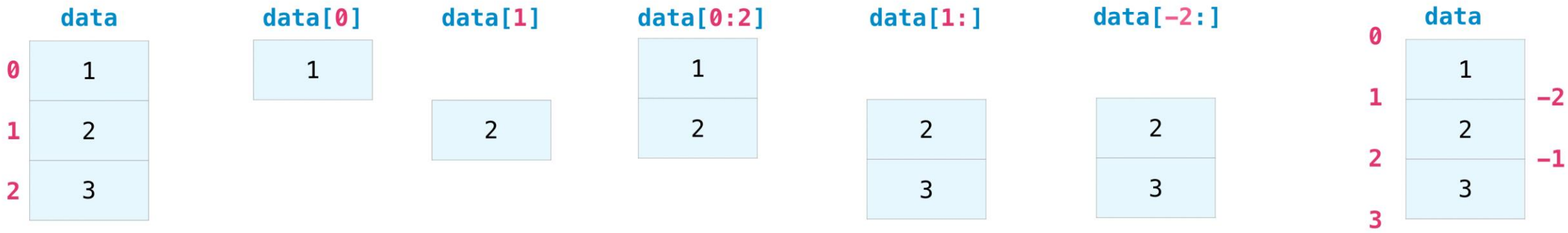


ARRAY OPERATIONS

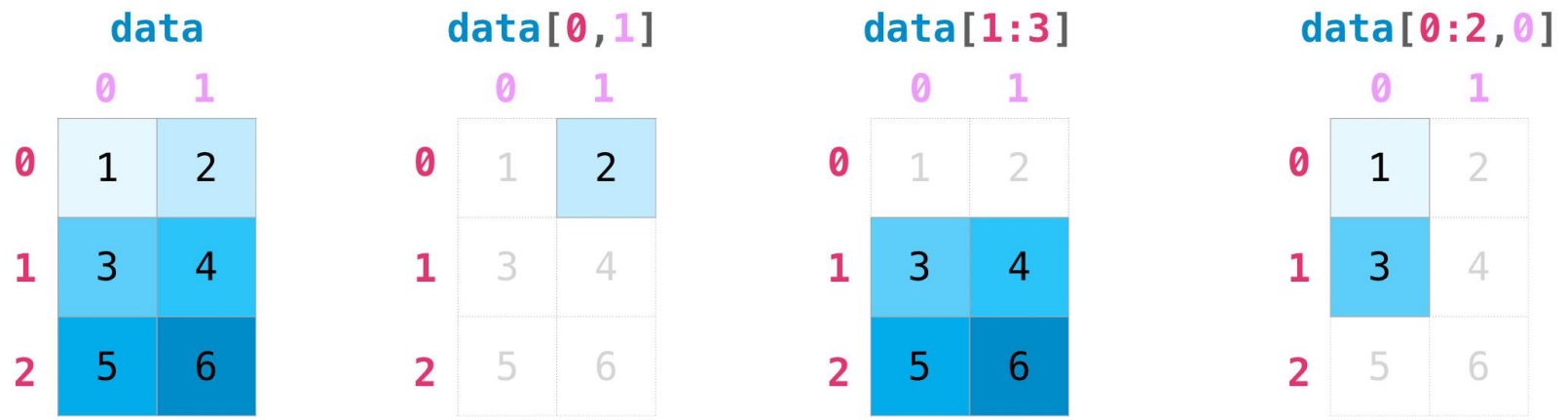




INDEXING & SLICING



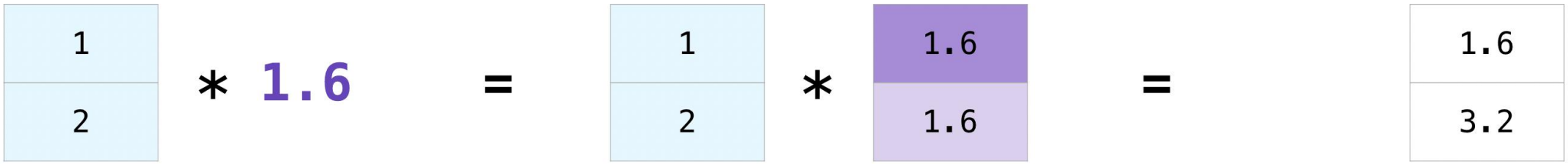
INDEXING & SLICING



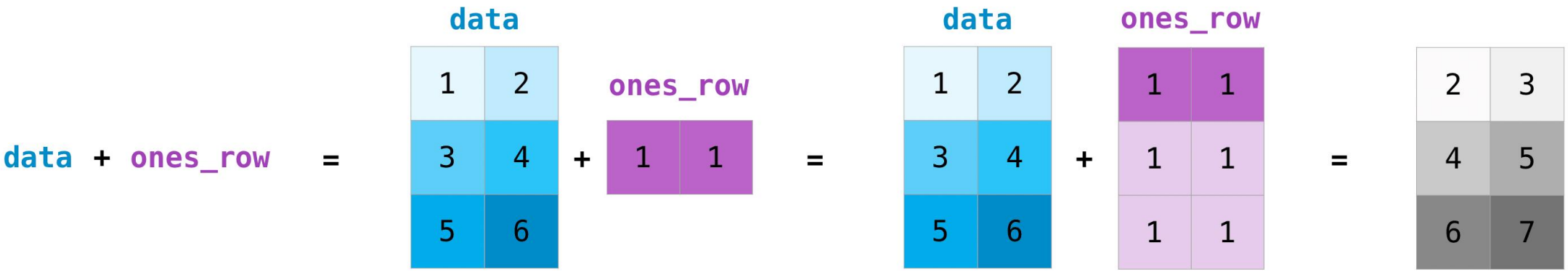
BOOLEAN INDEXING

```
>>> x = np.arange(35).reshape(5, 7)
>>> b = x > 20
>>> b[:, 5]
array([False, False, False,  True,  True])
>>> x[b[:, 5]]
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

BROADCASTING



BROADCASTING



3 - PYTORCH INTRODUCTION

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

DEEP LEARNING FRAMEWORKS

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI ICADE CIHS

PyTorch

HOW TO INSTALL IT

INSTALL PYTORCH

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	Stable (2.1.1)	Preview (Nightly)		
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 11.8	CUDA 12.1	ROCm 5.6	CPU
Run this Command:	`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118`			

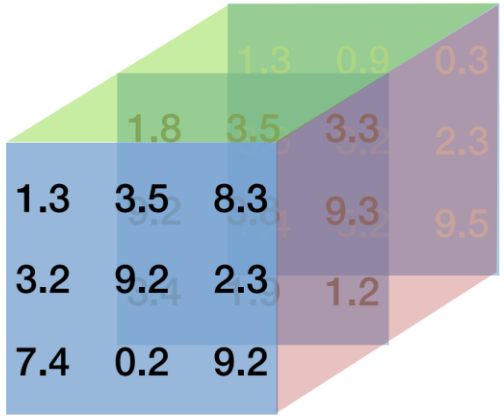
NOTE: PyTorch LTS has been deprecated. For more information, see this blog.

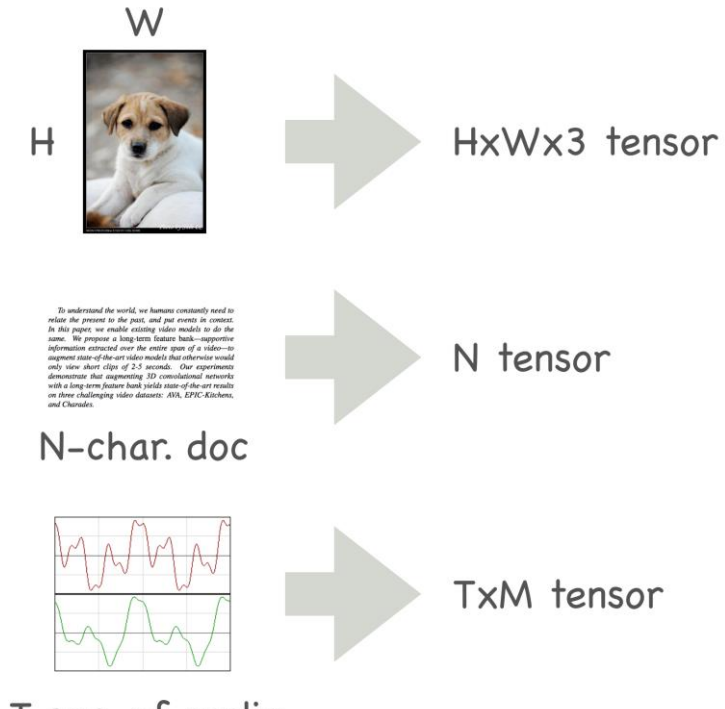
PYTORCH TENSOR

```
mytensor = torch.ones((3, 3))
mytensor = torch.zeros((3, 3), dtype=torch.long)
mytensor = torch.rand((3, 3), requires_grad=True)
mytensor = torch.empty((3, 3))
```

1.2	3.2	5.8

1.3	3.5	8.3
3.2	9.2	2.3
7.4	0.2	9.2





SHAPE OF TENSOR

We can access a tensor's *shape* (the length along each axis) by inspecting its `shape` attribute. Because we are dealing with a vector here, the `shape` contains just a single element and is identical to the size.

```
x.shape
```

```
torch.Size([12])
```

We can change the shape of a tensor without altering its size or values, by invoking `reshape`. For example, we can transform our vector `x` whose shape is (12,) to a matrix `X` with shape (3, 4). This new tensor retains all elements but reconfigures them into a matrix. Notice that the elements of our vector are laid out one row at a time and thus `x[3] == X[0, 3]`.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

INIT TENSORS

Practitioners often need to work with tensors initialized to contain all 0s or 1s. We can construct a tensor with all elements set to 0 and a shape of (2, 3, 4) via the `zeros` function.

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Similarly, we can create a tensor with all 1s by invoking `ones`.

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

LOADING FROM NUMPY

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

TO NUMPY

INDEXING & SLICING

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Beyond reading them, we can also *write* elements of a matrix by specifying indices.

```
X[1, 2] = 17
X
```

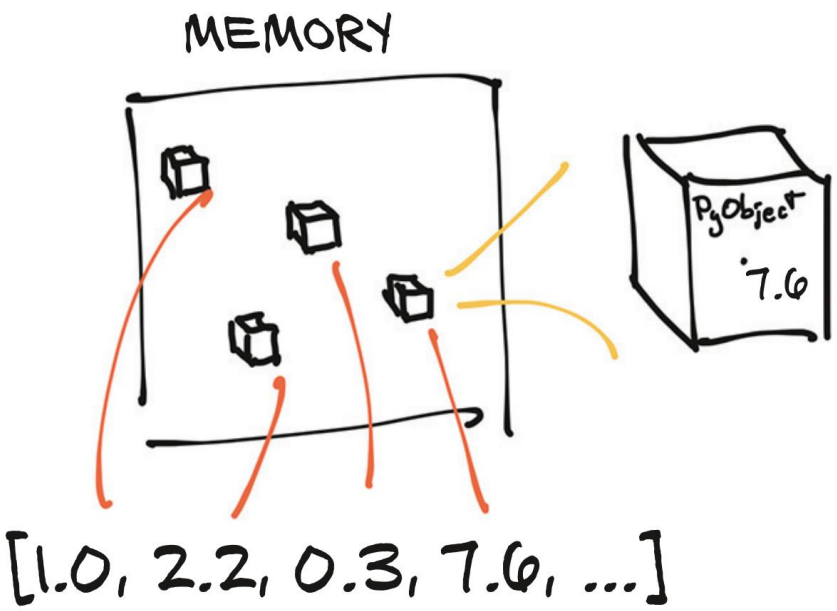
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

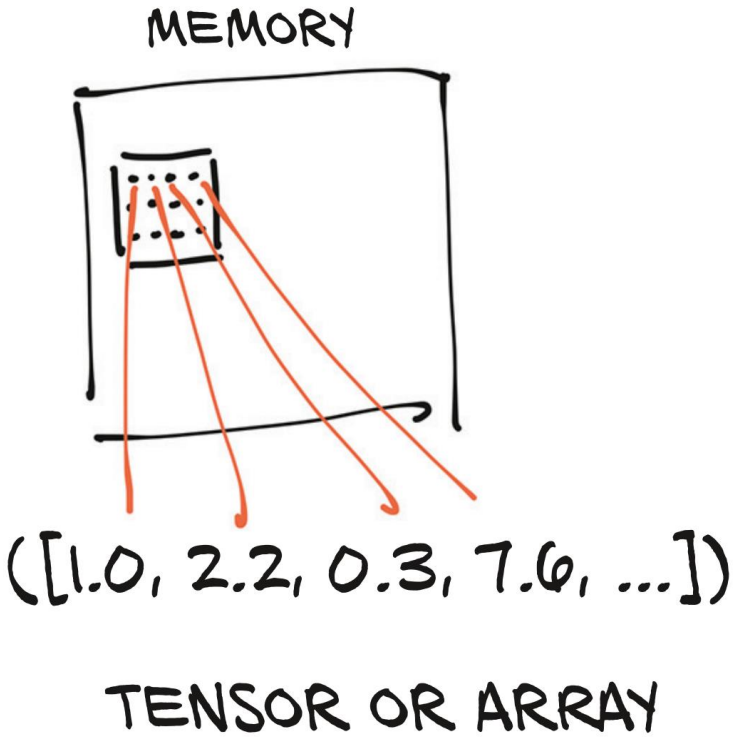
If we want to assign multiple elements the same value, we apply the indexing on the left-hand side of the assignment operation. For instance, `[:2, :]` accesses the first and second rows, where `:` takes all the elements along axis 1 (column). While we discussed indexing for matrices, this also works for vectors and for tensors of more than two dimensions.

```
X[:2, :] = 12
X
```

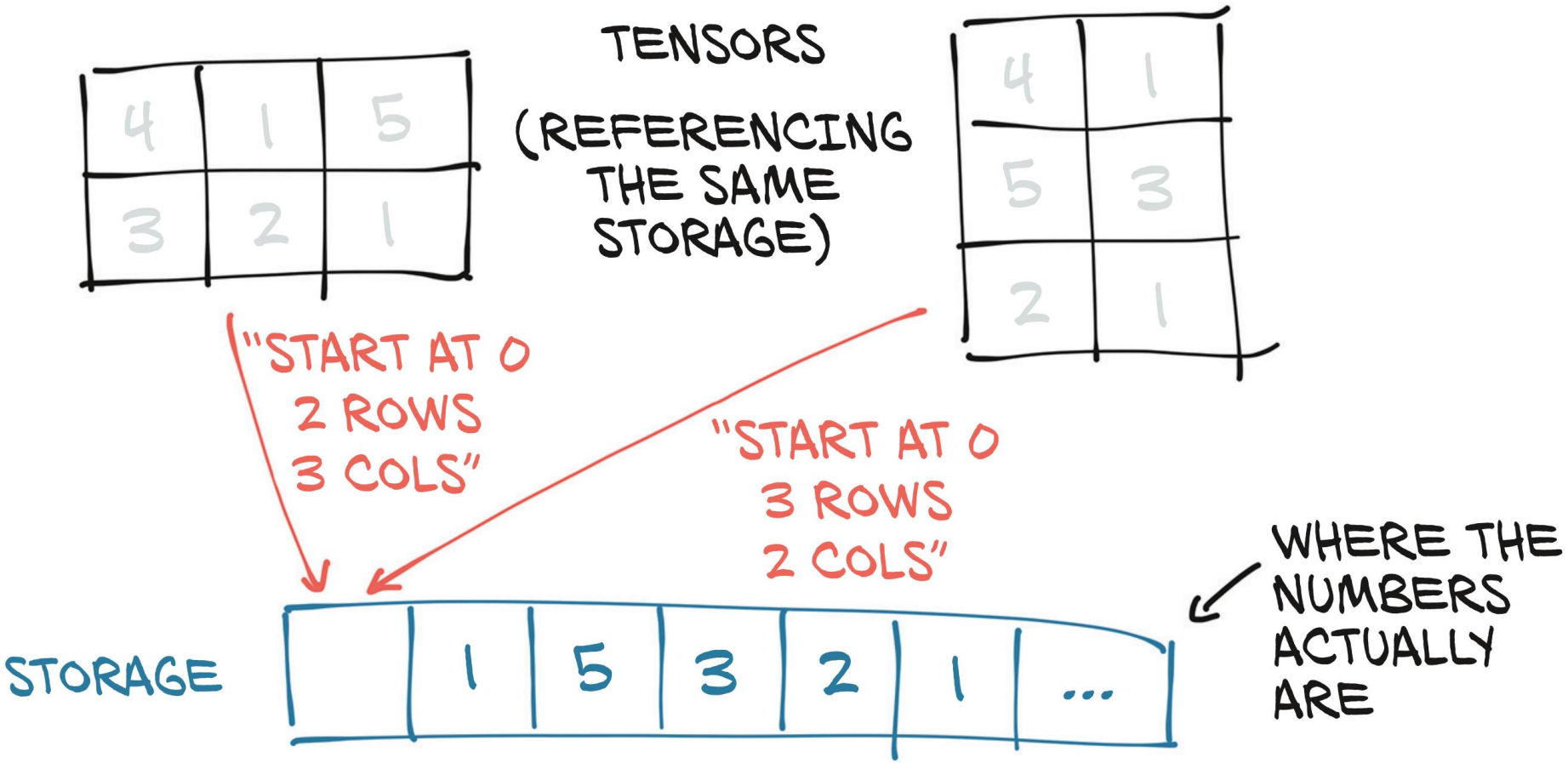
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

TENSOR/ARRAYS VS LIST





TENSOR STORAGE



VIEW VS CLONE

TORCH.TENSOR.VIEW

Tensor.view(*shape) → Tensor

Returns a new tensor with the same data as the `self` tensor but of a different `shape`.

The returned tensor shares the same data and must have the same number of elements, but may have a different size. For a tensor to be viewed, the new view size must be compatible with its original size and stride, i.e., each new view dimension must either be a subspace of an original dimension, or only span across original dimensions $d, d+1, \ldots, d+k$ that satisfy the following contiguity-like condition that $\forall i = d, \ldots, d+k-1$,

$$\text{stride}[i] = \text{stride}[i+1] \times \text{size}[i+1]$$

Otherwise, it will not be possible to view `self` tensor as `shape` without copying it (e.g., via `contiguous()`). When it is unclear whether a `view()` can be performed, it is advisable to use `reshape()`, which returns a view if the shapes are compatible, and copies (equivalent to calling `contiguous()`) otherwise.

Parameters

shape (*torch.Size* or *int*...) – the desired size

Example:

```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
>>> z.size()
torch.Size([2, 8])

>>> a = torch.randn(1, 2, 3, 4)
>>> a.size()
torch.Size([1, 2, 3, 4])
>>> b = a.transpose(1, 2)  # Swaps 2nd and 3rd dimension
>>> b.size()
torch.Size([1, 3, 2, 4])
>>> c = a.view(1, 3, 2, 4)  # Does not change tensor layout in memory
>>> c.size()
torch.Size([1, 3, 2, 4])
>>> torch.equal(b, c)
False
```

TORCH.CLONE

torch.clone(*input*, *, *memory_format=torch.preserve_format*) → Tensor

Returns a copy of `input`.

• NOTE

This function is differentiable, so gradients will flow back from the result of this operation to `input`. To create a tensor without an autograd relationship to `input` see `detach()`.

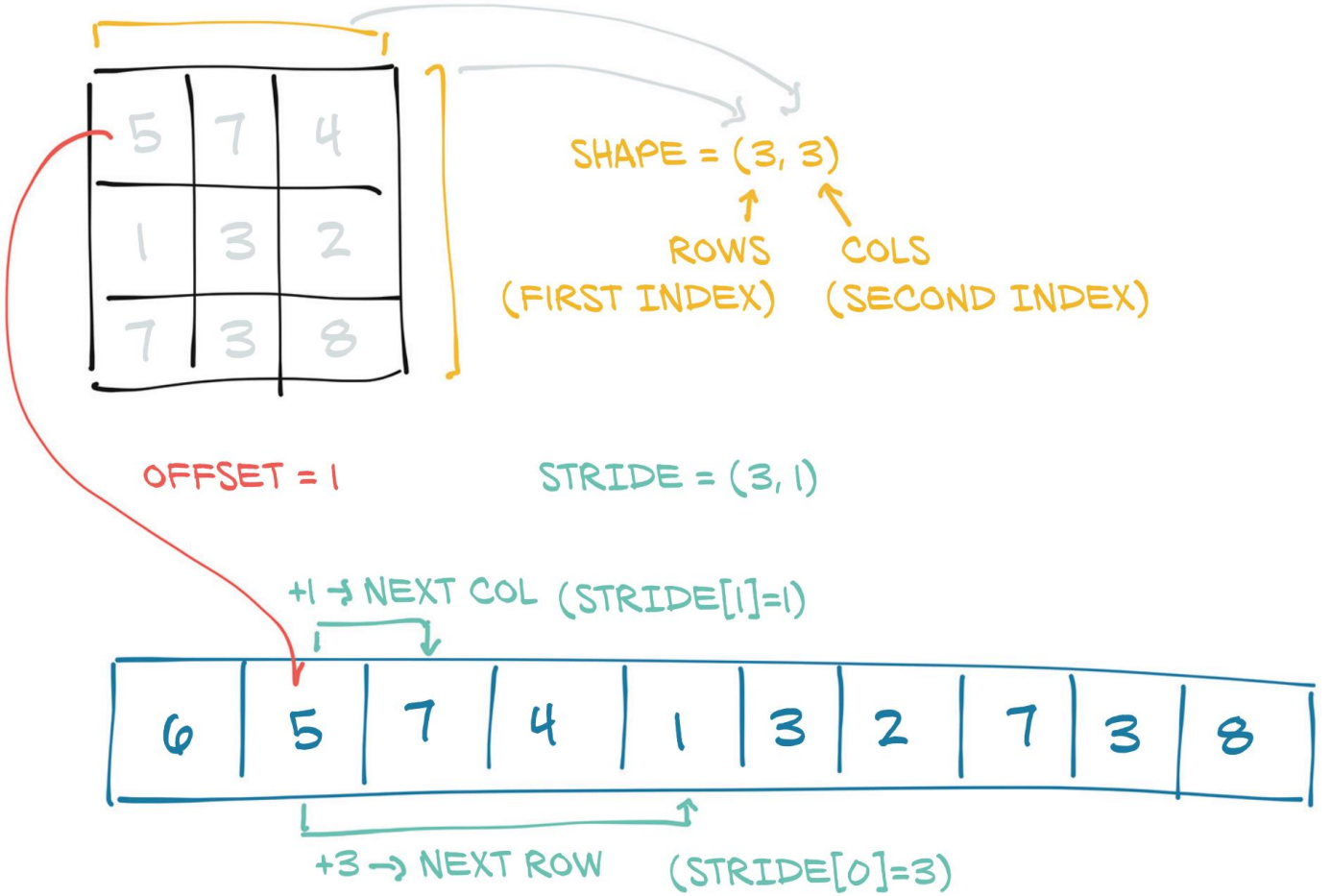
Parameters

input (*Tensor*) – the input tensor.

Keyword Arguments

memory_format (`torch.memory_format`, optional) – the desired memory format of returned tensor. Default: `torch.preserve_format`.

STORAGE, OFFSET AND STRIDE



CONTIGUOUS TENSOR

```
a = torch.rand((3, 3), requires_grad=True)
b = a.T
c = b.view(9)
>>>
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[3], line 3
      1 a = torch.rand((3, 3), requires_grad=True)
      2 b = a.T
----> 3 c = b.view(9)

RuntimeError: view size is not compatible with input tensor's size and stride (at least one dimension spans across two
contiguous subspaces). Use .reshape(...) instead.
```

BASIC SLICES ARE VIEWS

```
a = torch.ones((3, 3))
b = a[:, 0]
b[0] = 0
a
>>>
tensor([[0., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

ADVANCED SLICES ARE COPIES

```
a = torch.ones(10)
b = torch.tensor([0, 3, 4])
c = a[b]
c[0] = 0
a
>>> tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

IN-PLACE OPERATIONS

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

LOADING FROM NUMPY

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

FROM_NUMPY VS AS_TENSOR VS TENSOR

CLASS torch.Tensor

There are a few main ways to create a tensor, depending on your use case.

- To create a tensor with pre-existing data, use `torch.tensor()`.
- To create a tensor with specific size, use `torch.*` tensor creation ops (see Creation Ops).
- To create a tensor with the same size (and similar types) as another tensor, use `torch.*_like` tensor creation ops (see Creation Ops).
- To create a tensor with similar type but different size as another tensor, use `tensor.new_*` creation ops.
- There is a legacy constructor `torch.Tensor` whose use is discouraged. Use `torch.tensor()` instead.

Tensor.__init__(*self*, *data*)

This constructor is deprecated, we recommend using `torch.tensor()` instead. What this constructor does depends on the type of `data`.

- If `data` is a Tensor, returns an alias to the original Tensor. Unlike `torch.tensor()`, this tracks autograd and will propagate gradients to the original Tensor. `device` kwarg is not supported for this `data` type.
- If `data` is a sequence or nested sequence, create a tensor of the default dtype (typically `torch.float32`) whose data is the values in the sequences, performing coercions if necessary. Notably, this differs from `torch.tensor()` in that this constructor will always construct a float tensor, even if the inputs are all integers.
- If `data` is a `torch.Size`, returns an empty tensor of that size.

This constructor does not support explicitly specifying `dtype` or `device` of the returned tensor. We recommend using `torch.tensor()` which provides this functionality.

Args:

data (array_like): The tensor to construct from.

Keyword args:

device (`torch.device`, optional): the desired device of returned tensor.

Default: if None, same `torch.device` as this tensor.

• SEE ALSO

`torch.tensor()` never shares its data and creates a new “leaf tensor” (see Autograd mechanics).

• SEE ALSO

`torch.as_tensor()` preserves autograd history and avoids copies where possible. `torch.from_numpy()` creates a tensor that shares storage with a NumPy array.

TENSOR TO FLOAT

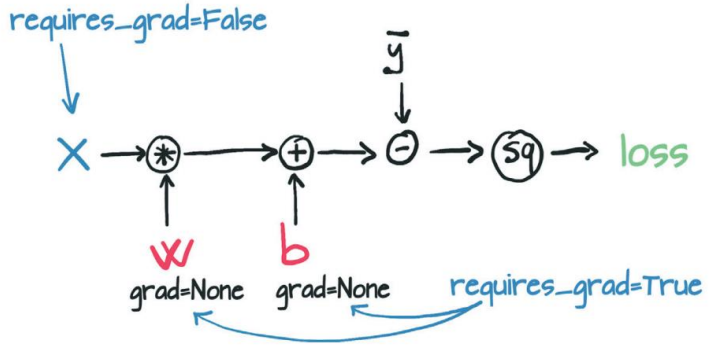
To convert a size-1 tensor to a Python scalar, we can invoke the `item` function or Python's built-in functions.

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

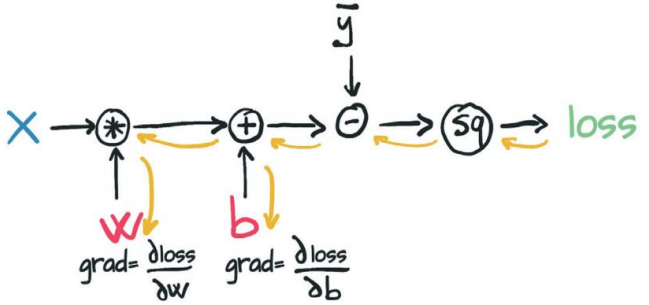
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

4 - AUTOGRAD

AUTODIFF - AUTOGRAD

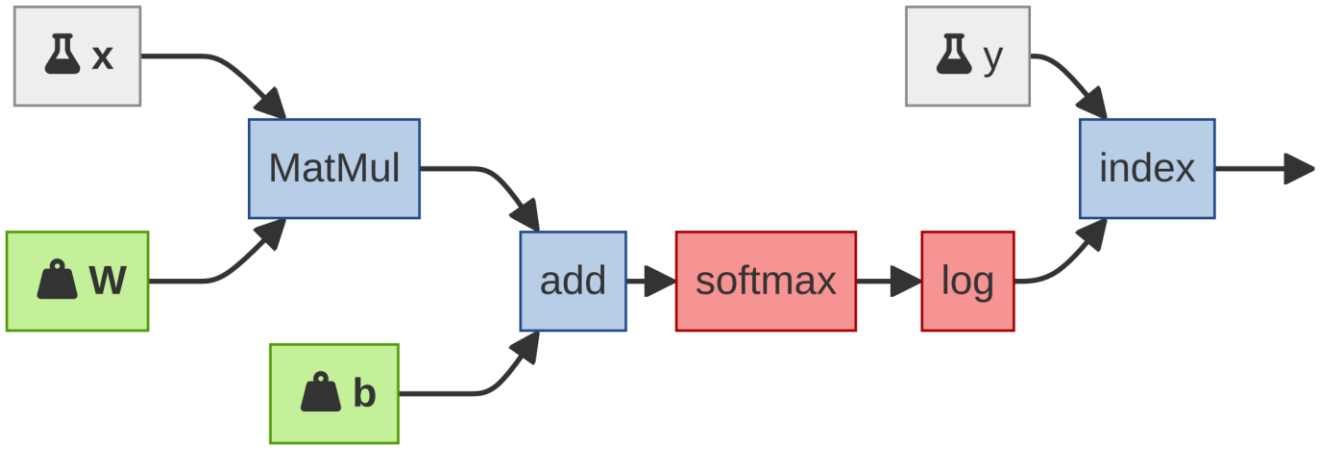


loss.backward()



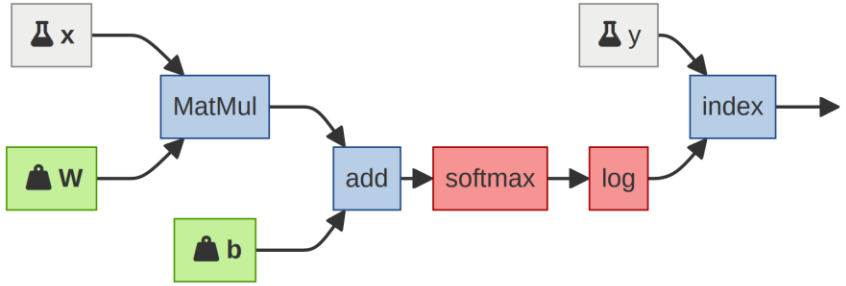
COMPUTATIONAL GRAPH

$$
\begin{aligned}
l(\theta|\mathbf{x}, \mathbf{y}) &= \log\left(\text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})\right)_y \\
&= \text{index}\left(\log\left(\text{softmax}\left(\text{add}\left(\text{matmul}(\mathbf{W}, \mathbf{x}), \mathbf{b}\right)\right)\right), y\right)
\end{aligned}
$$

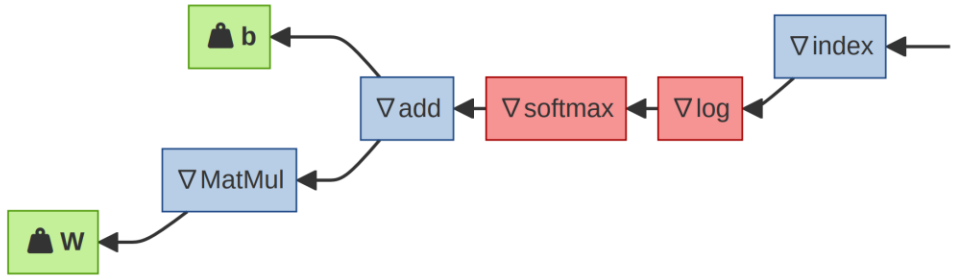


GRADIENTS ON COMPUTATION GRAPHS

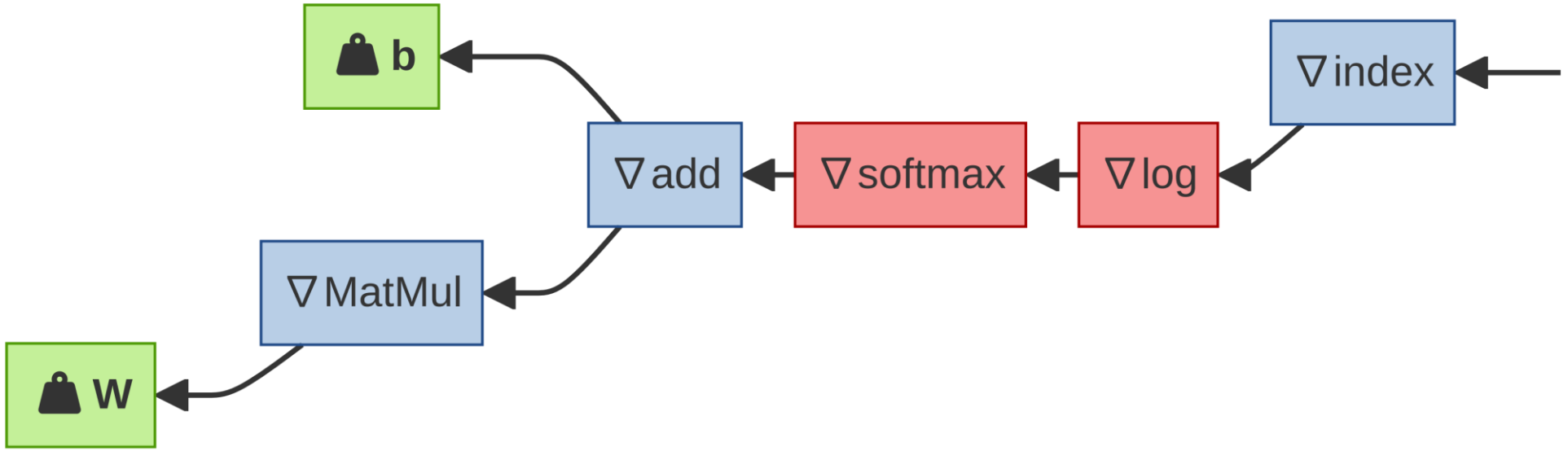
$$l(\theta|\mathbf{x},\mathbf{y}) = \text{index}\left(\log\left(\text{softmax}\left(\text{add}\left(\text{matmul}(\mathbf{W},\mathbf{x}),\mathbf{b}\right)\right)\right),y\right)$$



$$\nabla_\theta l(\theta|\mathbf{x},\mathbf{y}) = \nabla\text{index}(\ldots)\nabla\log(\ldots)\nabla\text{softmax}(\ldots)\nabla\text{add}(\ldots)\left(\nabla\text{matmul}(\mathbf{W},\mathbf{x})\nabla_\theta\mathbf{W} + \nabla_\theta\mathbf{b}\right)$$



BACKWARD PASS



BACKWARD

```
input = torch.randn(1, 1, 28, 28)
out = net(input)
print(out.size())

target = torch.tensor([3], dtype=torch.long)
loss_fn = nn.CrossEntropyLoss()  # LogSoftmax + ClassNLL Loss
err = loss_fn(out, target)
err.backward()
```

TORCH.TENSOR.BACKWARD

Tensor.backward(*gradient=None*, *retain_graph=None*, *create_graph=False*, *inputs=None*)[source]

Computes the gradient of current tensor wrt graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying `gradient`. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self`.

This function accumulates gradients in the leaves - you might need to zero `.grad` attributes or set them to `None` before calling it. See Default gradient layouts for details on the memory layout of accumulated gradients.

NOTE

If you run any forward ops, create `gradient`, and/or call `backward` in a user-specified CUDA stream context, see Stream semantics of backward passes.

NOTE

When `inputs` are provided and a given input is not a leaf, the current implementation will call its grad_fn (though it is not strictly needed to get this gradients). It is an implementation detail on which the user should not rely. See https://github.com/pytorch/pytorch/pull/60521#issuecomment-867061780 for more details.

Parameters

- **gradient** (*Tensor or None*) – Gradient w.r.t. the tensor. If it is a tensor, it will be automatically converted to a Tensor that does not require grad unless `create_graph` is True. None values can be specified for scalar Tensors or ones that don't require grad. If a None value would be acceptable then this argument is optional.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grads will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to `False`.

AUTOMATIC DIFF OF TENSOR

```
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The gradient is None by default
```

AUTOMATIC DIFF OF TENSOR

We now calculate our function of x and assign the result to y.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

We can now take the gradient of y with respect to x by calling its backward method. Next, we can access the gradient via x's grad attribute.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

RESET GRADIENTS

Now let's calculate another function of `x` and take its gradient. Note that PyTorch does not automatically reset the gradient buffer when we record a new gradient. Instead, the new gradient is added to the already-stored gradient. This behavior comes in handy when we want to optimize the sum of multiple objective functions. To reset the gradient buffer, we can call `x.grad.zero_()` as follows:

```
x.grad.zero_()  # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

BACKWARD FOR NON-SCALAR VARIABLES

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

DETACH

TORCH.TENSOR.DETACH

`Tensor.detach()`

Returns a new Tensor, detached from the current graph.

The result will never require gradient.

This method also affects forward mode AD gradients and the result will never have forward mode AD gradients.

NOTE

Returned Tensor shares the same storage with the original one. In-place modifications on either of them will be seen, and may trigger errors in correctness checks. IMPORTANT NOTE: Previously, in-place size / stride / storage changes (such as *resize_* / *resize_as_* / *set_* / *transpose_*) to the returned tensor also update the original tensor. Now, these in-place changes will not update the original tensor anymore, and will instead trigger an error. For sparse tensors: In-place indices / values changes (such as *zero_* / *copy_* / *add_*) to the returned tensor will not update the original tensor anymore, and will instead trigger an error.

HOW IS COMPUTATIONAL GRAPH RECORDED?

```
c = 2 * b
print(c)

d = c + 1
print(d)
```

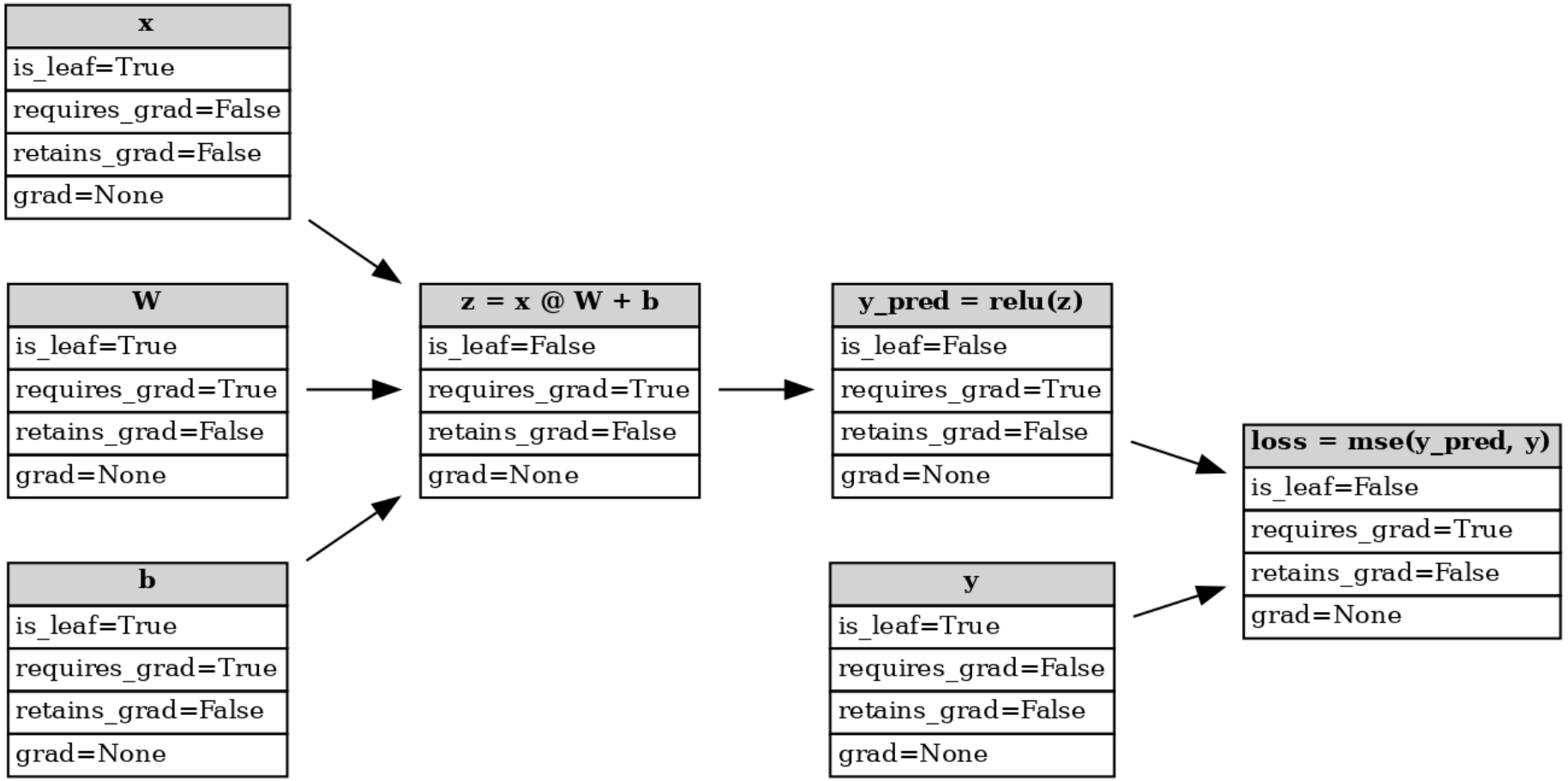
Out:

```
tensor([ 0.0000e+00,  5.1764e-01,  1.0000e+00,  1.4142e+00,  1.7321e+00,
         1.9319e+00,  2.0000e+00,  1.9319e+00,  1.7321e+00,  1.4142e+00,
         1.0000e+00,  5.1764e-01, -1.7485e-07, -5.1764e-01, -1.0000e+00,
        -1.4142e+00, -1.7321e+00, -1.9319e+00, -2.0000e+00, -1.9319e+00,
        -1.7321e+00, -1.4142e+00, -1.0000e+00, -5.1764e-01,  3.4969e-07],
       grad_fn=<MulBackward0>)
tensor([ 1.0000e+00,  1.5176e+00,  2.0000e+00,  2.4142e+00,  2.7321e+00,
         2.9319e+00,  3.0000e+00,  2.9319e+00,  2.7321e+00,  2.4142e+00,
         2.0000e+00,  1.5176e+00,  1.0000e+00,  4.8236e-01, -3.5763e-07,
        -4.1421e-01, -7.3205e-01, -9.3185e-01, -1.0000e+00, -9.3185e-01,
        -7.3205e-01, -4.1421e-01,  4.7684e-07,  4.8236e-01,  1.0000e+00],
       grad_fn=<AddBackward0>)
```

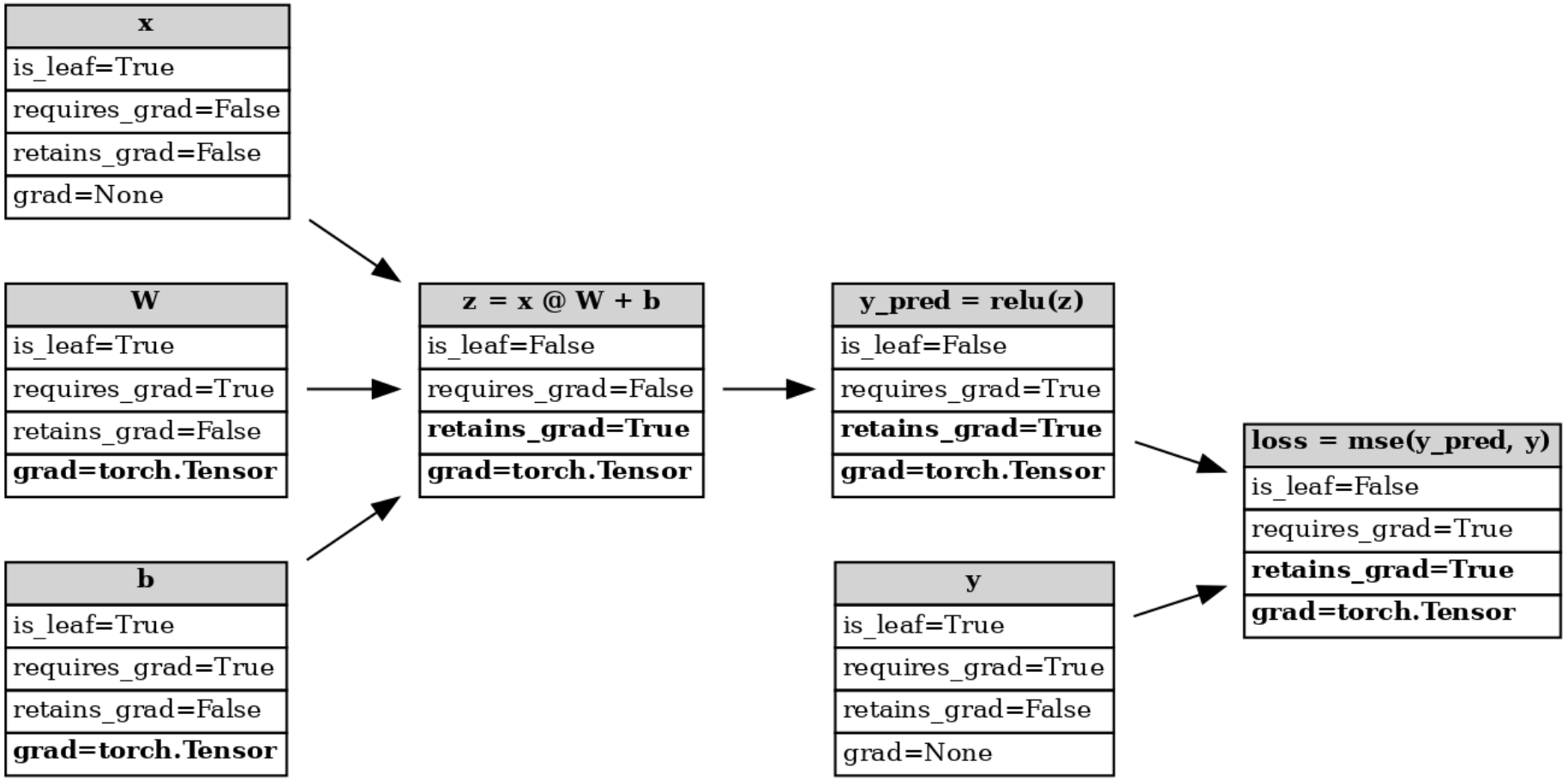
GRAD FN

```
import torch
a = torch.randn((3,3), requires_grad = True)
w1 = torch.randn((3,3), requires_grad = True)
w2 = torch.randn((3,3), requires_grad = True)
w3 = torch.randn((3,3), requires_grad = True)
w4 = torch.randn((3,3), requires_grad = True)
b = w1*a
c = w2*a
d = w3*b + w4*c
L = 10 - d
print("The grad fn for a is", a.grad_fn)
print("The grad fn for d is", d.grad_fn)

>>> The grad fn for a is None
>>> The grad fn for d is <AddBackward0 object at 0x1033afe48>
```

LEAF TENSORS



LEAF TENSORS



AUTOGRAD AND IN-PLACE OPERATIONS

In every example in this notebook so far, we've used variables to capture the intermediate values of a computation. Autograd needs these intermediate values to perform gradient computations. *For this reason, you must be careful about using in-place operations when using autograd.* Doing so can destroy information you need to compute derivatives in the `backward()` call. PyTorch will even stop you if you attempt an in-place operation on leaf variable that requires autograd, as shown below.

• NOTE

The following code cell throws a runtime error. This is expected.

```
a = torch.linspace(0., 2. * math.pi, steps=25, requires_grad=True)
torch.sin_(a)
```

INPLACE WHEN VARIABLE NEEDED

```
a = torch.randn(3, requires_grad=True)
b = 2 * a
c = b ** 2  # replace this with c = b + 2 and the autograd error will go away
b += 1  # inplace operation!
c.sum().backward()

>>>
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[23], line 5
      3 c = b ** 2  # replace this with c = b + 2 and the autograd error will go away
      4 b += 1  # inplace operation!
----> 5 c.sum().backward()

RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation:
[torch.FloatTensor [5]], which is output 0 of AddBackward0, is at version 1; expected version 0 instead. Hint: enable
anomaly detection to find the operation that failed to compute its gradient, with
torch.autograd.set_detect_anomaly(True).
```

AUTOGRAD AND INPLACE OPERATIONS

```
a = torch.rand(3).requires_grad_(True)
a += 1
b = a * 2
c = b * 4
c.sum().backward()
b.grad
```

```
>>> UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute
won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf
Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access
the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at
aten/src/ATen/core/TensorBody.h:489.)
  b.grad
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[5], line 2
      1 a = torch.rand(3).requires_grad_(True)
----> 2 a += 1
      3 b = a * 2
      4 c = b * 4

RuntimeError: a leaf Variable that requires grad is being used in an in-place operation.
```

INTERMEDIATE GRADIENTS

```
a = torch.rand(3).requires_grad_(True)
b = a * 2
c = b * 4
c.sum().backward()
b.grad

>>> UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute
won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf
Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access
the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at
aten/src/ATen/core/TensorBody.h:489.)
```

DEACTIVATE INTERMEDIATE GRADIENTS

```
a = torch.rand(3).requires_grad_(True)
b = a * 2
c = b * 4
b.requires_grad_(False)
c.sum().backward()
b.grad
```

```
>>> UserWarning: The .grad attribute of a Tensor that is not a leaf Tensor is being accessed. Its .grad attribute
won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf
Tensor, use .retain_grad() on the non-leaf Tensor. If you access the non-leaf Tensor by mistake, make sure you access
the leaf Tensor instead. See github.com/pytorch/pytorch/pull/30531 for more informations. (Triggered internally at
aten/src/ATen/core/TensorBody.h:489.)
  b.grad
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[4], line 4
      2 b = a * 2
      3 c = b * 4
----> 4 b.requires_grad_(False)
      5 c.sum().backward()
      6 b.grad

RuntimeError: you can only change requires_grad flags of leaf variables. If you want to use a computed variable in a
subgraph that doesn't require differentiation use var_no_grad = var.detach().
```

CUSTOM AUTOGRAD FUNCTIONS

CLASS torch.autograd.Function(*args, **kwargs) [SOURCE]

Base class to create custom *autograd.Function*.

To create a custom *autograd.Function*, subclass this class and implement the `forward()` and `backward()` static methods. Then, to use your custom op in the forward pass, call the class method `apply`. Do not call `forward()` directly.

To ensure correctness and best performance, make sure you are calling the correct methods on `ctx` and validating your backward function using `torch.autograd.gradcheck()`.

See Extending torch.autograd for more details on how to use this class.

Examples:

```
>>> class Exp(Function):
>>>     @staticmethod
>>>     def forward(ctx, i):
>>>         result = i.exp()
>>>         ctx.save_for_backward(result)
>>>         return result
>>>
>>>     @staticmethod
>>>     def backward(ctx, grad_output):
>>>         result, = ctx.saved_tensors
>>>         return grad_output * result
>>>
>>> # Use it by calling the apply method:
>>> output = Exp.apply(input)
```

5 - PYTORCH API

BASIC BLOCK FOR NEURAL NETWORKS

```python
class MyModel(torch.nn.Module):

    def __init__(self, *args) -> None:

        # TODO

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:

        # TODO
```

MODULE

CLASS torch.nn.Module(**args*, ***kwargs*) [SOURCE]

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call `to()`, etc.

• NOTE

As per the example above, an `__init__()` call to the parent class must be made before assignment on the child.

CUSTOM MODULE

```
class MLP(nn.Module):
    def __init__(self):
        # Call the constructor of the parent class nn.Module to perform
        # the necessary initialization
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.out = nn.LazyLinear(10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

SEQUENTIAL MODULE

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

FORWARD METHOD

```python
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Random weight parameters that will not compute gradients and
        # therefore keep constant during training
        self.rand_weight = torch.rand((20, 20))
        self.linear = nn.LazyLinear(20)

    def forward(self, X):
        X = self.linear(X)
        X = F.relu(X @ self.rand_weight + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.linear(X)
        # Control flow
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

PARAMETER ACCESS - STATE DICT

```
net[2].state_dict()
```

```
OrderedDict([('weight',
              tensor([[-0.1649,  0.0605,  0.1694, -0.2524,  0.3526, -0.3414, -
↪0.2322,  0.0822]])),
             ('bias', tensor([0.0709]))])
```

PARAMETER ACCESS - ALL PARAMETERS AT ONCE

```
[(name, param.shape) for name, param in net.named_parameters()]
```

```
[('0.weight', torch.Size([8, 4])),
 ('0.bias', torch.Size([8])),
 ('2.weight', torch.Size([1, 8])),
 ('2.bias', torch.Size([1]))]
```

LAZY INITIALIZATION

To begin, let's instantiate an MLP.

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
```

At this point, the network cannot possibly know the dimensions of the input layer's weights because the input dimension remains unknown.

Consequently the framework has not yet initialized any parameters. We confirm by attempting to access the parameters below.

```
net[0].weight
```

```
<UninitializedParameter>
```

Next let's pass data through the network to make the framework finally initialize parameters.

```
X = torch.rand(2, 20)
net(X)

net[0].weight.shape
```

```
torch.Size([256, 20])
```

MODEL PARAMETERS

PARAMETER

CLASS torch.nn.parameter.Parameter(*data=None*, *requires_grad=True*) [SOURCE]

A kind of Tensor that is to be considered a module parameter.

Parameters are `Tensor` subclasses, that have a very special property when used with `Module` s - when they're assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in `parameters()` iterator. Assigning a Tensor doesn't have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as `Parameter`, these temporaries would get registered too.

Parameters

- **data** (*Tensor*) – parameter tensor.
- **requires_grad** (*bool, optional*) – if the parameter requires gradient. Note that the torch.no_grad() context does NOT affect the default behavior of Parameter creation–the Parameter will still have *requires_grad=True* in `no_grad` mode. See Locally disabling gradient computation for more details. Default: *True*

LAYER WITH PARAMETERS

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))

    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

CHECK MODEL PARAMETERS

parameters(*recurse=True*) [SOURCE]

Return an iterator over module parameters.

This is typically passed to an optimizer.

Parameters

recurse (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields

Parameter – module parameter

Return type

Iterator[*Parameter*]

Example:

```
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

MODEL BUFFERS

register_buffer(*name*, *tensor*, *persistent=True*) [SOURCE]

Add a buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the module's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting `persistent` to `False`. The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this module's `state_dict`.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor or None*) – buffer to be registered. If `None`, then operations that run on buffers, such as `cuda`, are ignored. If `None`, the buffer is **not** included in the module's `state_dict`.
- **persistent** (*bool*) – whether the buffer is part of this module's `state_dict`.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

SAVE TENSORS

```
# Save to file
x = torch.tensor([0, 1, 2, 3, 4])
torch.save(x, "tensor.pt")
# Save to io.BytesIO buffer
buffer = io.BytesIO()
torch.save(x, buffer)
```

LOAD TENSORS

```
torch.load("tensors.pt", weights_only=True)
# Load all tensors onto the CPU
torch.load("tensors.pt", map_location=torch.device("cpu"), weights_only=True)
# Load all tensors onto the CPU, using a function
```

WHAT IS A STATE DICT?

```
Model's state_dict:
conv1.weight     torch.Size([6, 3, 5, 5])
conv1.bias   torch.Size([6])
conv2.weight     torch.Size([16, 6, 5, 5])
conv2.bias   torch.Size([16])
fc1.weight   torch.Size([120, 400])
fc1.bias     torch.Size([120])
fc2.weight   torch.Size([84, 120])
fc2.bias     torch.Size([84])
fc3.weight   torch.Size([10, 84])
fc3.bias     torch.Size([10])

Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.001, 'momentum': 0.9, 'dampening': 0, 'weight_decay': 0, 'nesterov': False,
'params': [4675713712, 4675713784, 4675714000, 4675714072, 4675714216, 4675714288, 4675714432,
4675714504, 4675714648, 4675714720]}]
```

SAVE WITH STATE DICT

Save/Load `state_dict` (Recommended)

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, weights_only=True))
model.eval()
```

SAVE ENTIRE MODEL

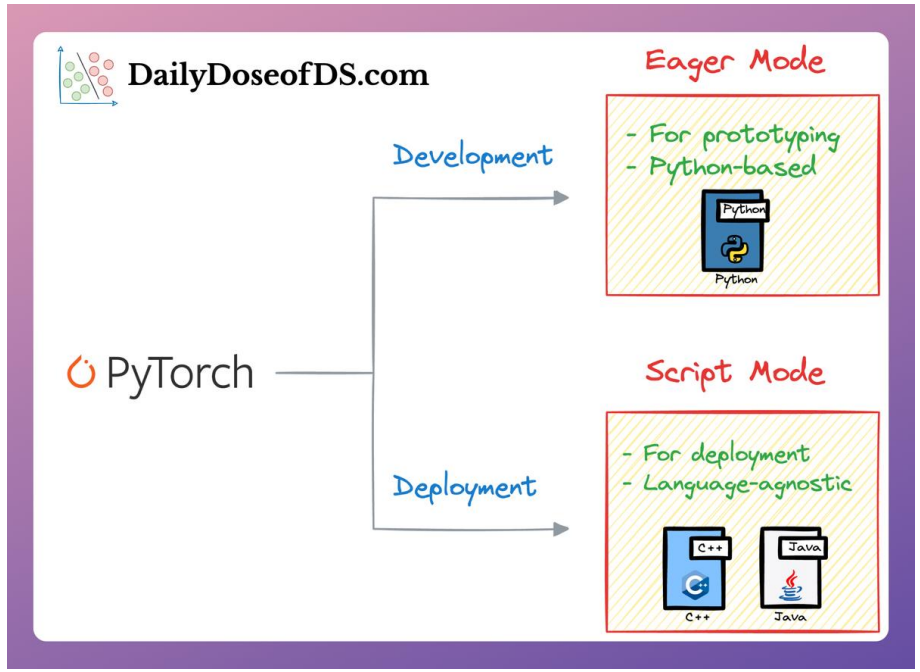
Save:

```
torch.save(model, PATH)
```

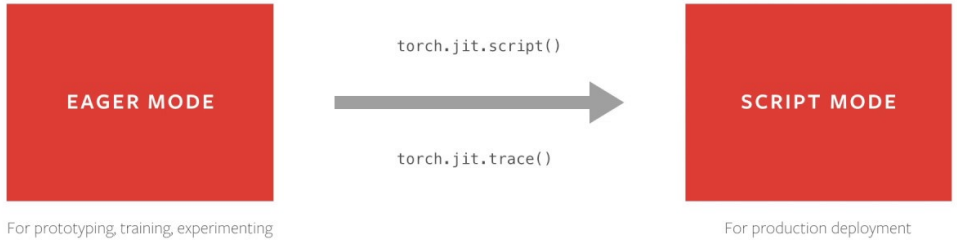
Load:

```
# Model class must be defined somewhere
model = torch.load(PATH, weights_only=False)
model.eval()
```

WHAT IS TORCHSCRIPT?



TOOLS TO TRANSITION FROM EAGER TO SCRIPT



SAVE WITH TORCHSCRIPT

Export:

```
model_scripted = torch.jit.script(model) # Export to TorchScript
model_scripted.save('model_scripted.pt') # Save
```

Load:

```
model = torch.jit.load('model_scripted.pt')
model.eval()
```

TORCH.ENABLE_GRAD()

```
with torch.enable_grad():
  # rest of the code
```

```
@torch.enable_grad()
def function(args):
  # rest of the code
```

TORCH.NO_GRAD()

```
with torch.no_grad():
  # rest of the code
```

```
@torch.no_grad()
def function(args):
  # rest of the code
```

TORCH.INFERENCE_MODE()

```
with torch.inference_mode():
  # rest of the code
```

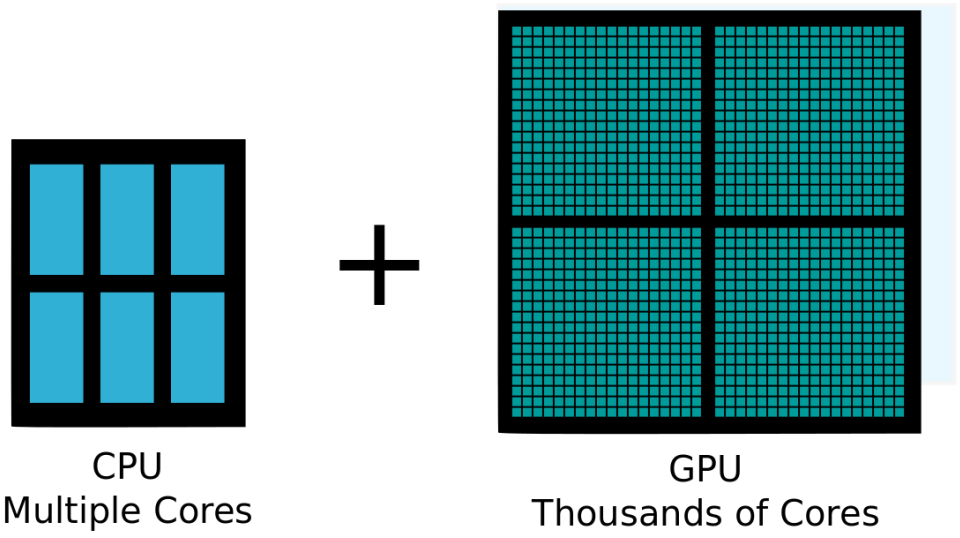
```
@torch.inference_mode()
def function(args):
  # rest of the code
```

COMPARISON BETWEEN MODES

Mode	Excludes operations from being recorded in backward graph	Skips additional autograd tracking overhead	Tensors created while the mode is enabled can be used in grad-mode later	Examples
default			✓	Forward pass
no-grad	✓		✓	Optimizer updates
inference	✓	✓		Data processing, model evaluation

MANAGING DEVICE

```
device = (
  torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
)
mytensor = mytensor.to(device)
```

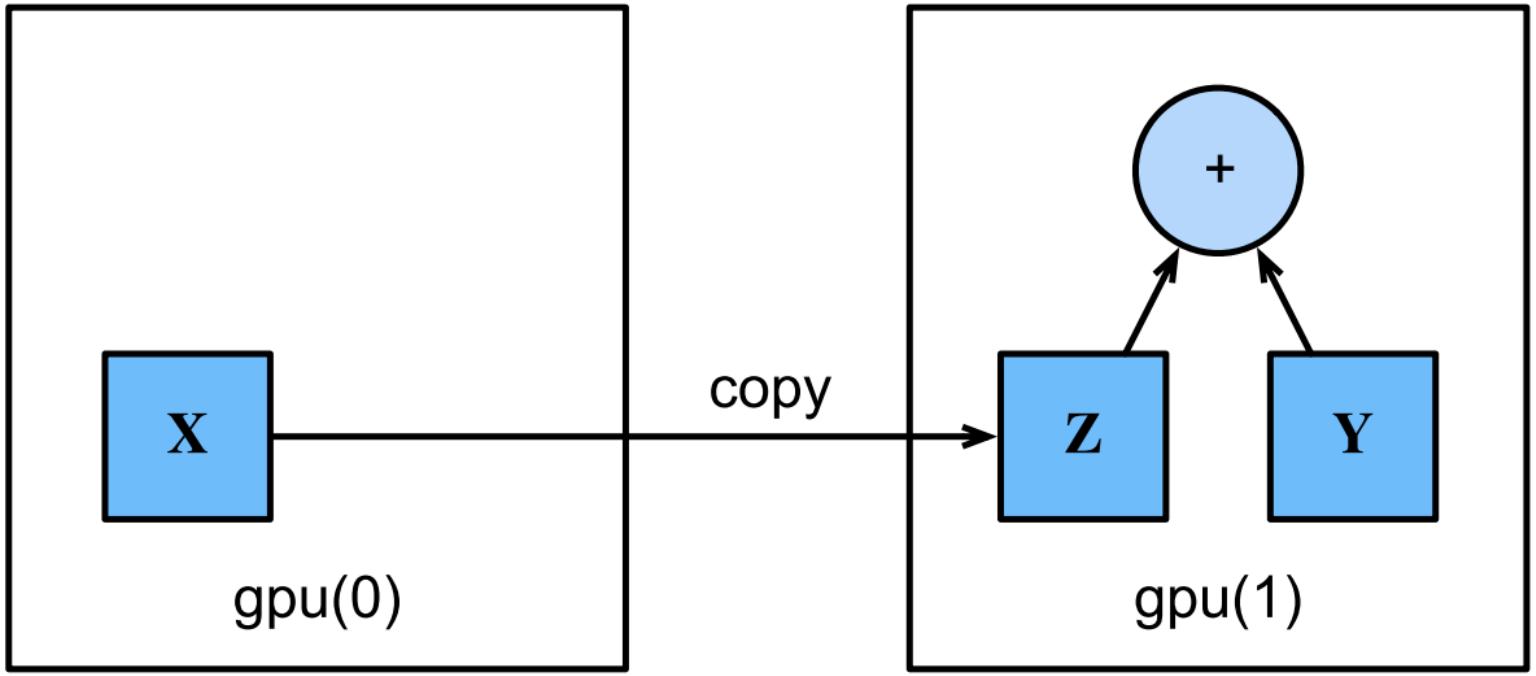


DEVICE ON TENSORS

```
x = torch.tensor([1, 2, 3])
x.device
```

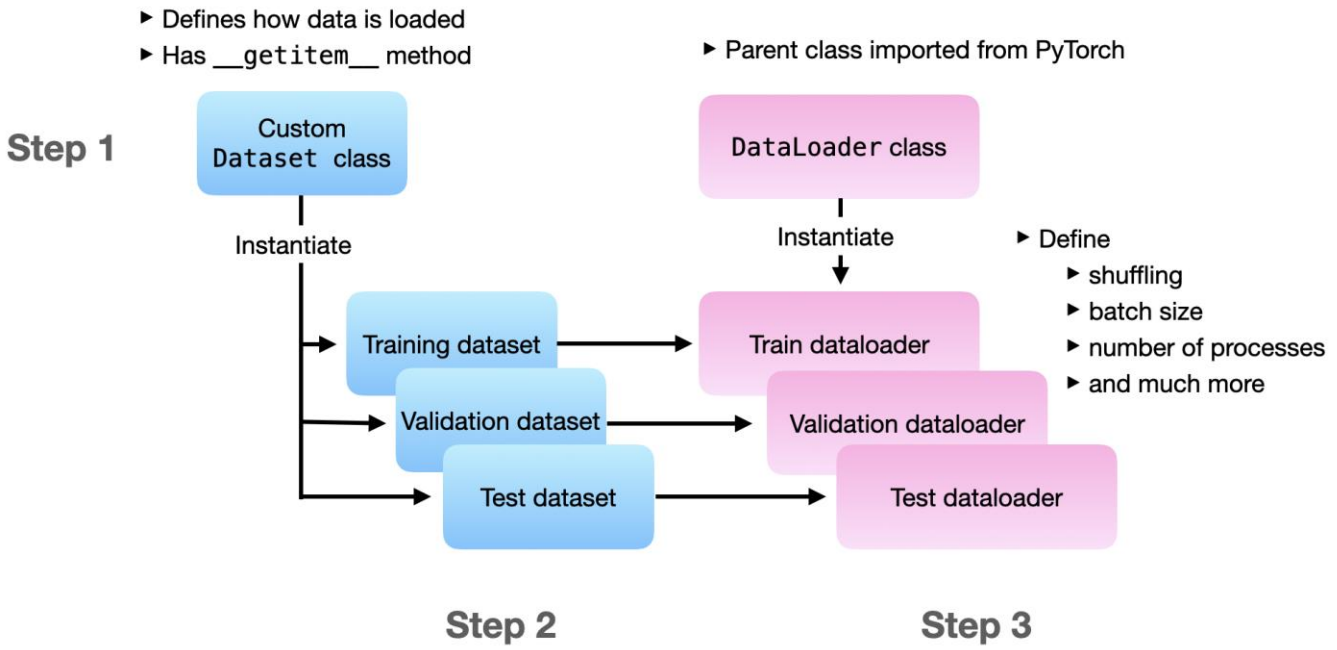
```
device(type='cpu')
```

GPU DEVICES



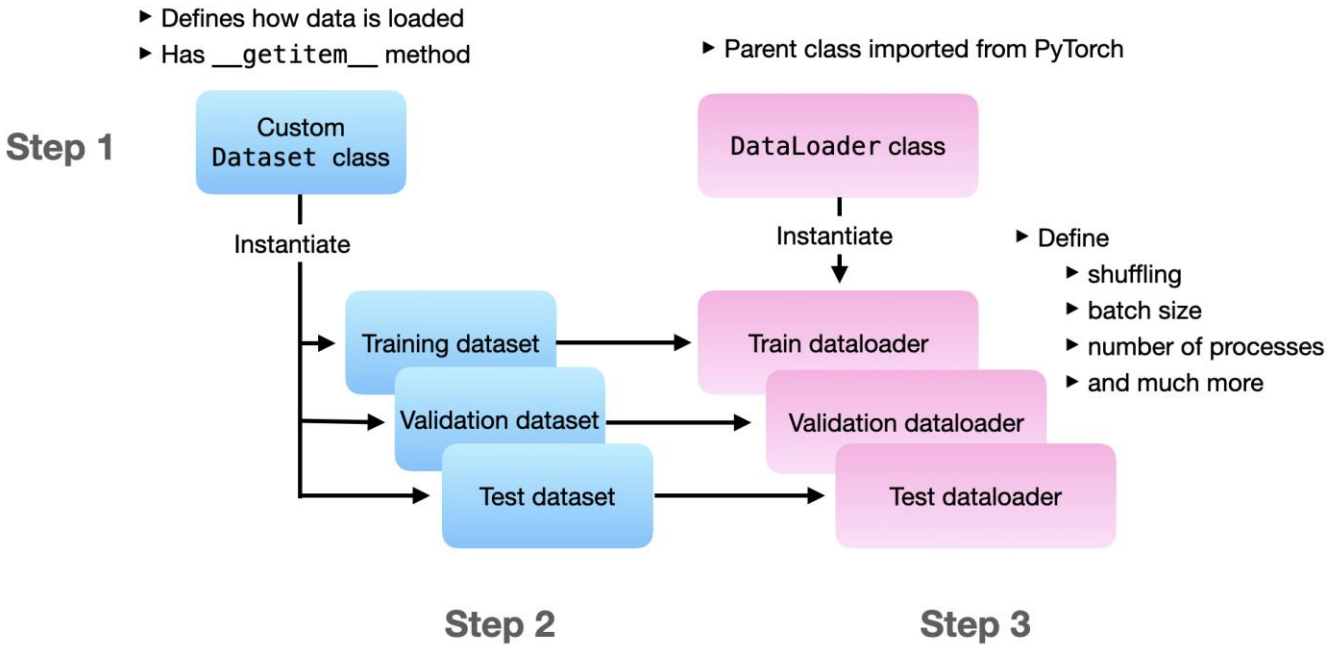
DATASET

```python
class MyDataset(Dataset):
    def __init__(self, inputs: list[list[float]], targets: List[int]) -> None:

        self.inputs = inputs
        self.targets = targets

    def __len__(self) -> int:
        return len(self.inputs)

    def __getitem__(self, index: int) -> tuple[torch.Tensor, int]:

        inputs: torch.Tensor = torch.tensor(self.inputs[index])
        target: int = self.targets[index]

        return inputs, target
```



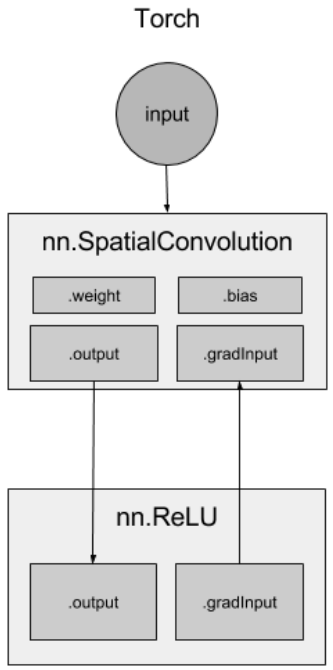
DATALOADER

```python
dataloader: DataLoader = DataLoader(
  dataset,
  batch_size=64,
  shuffle=True,
  num_workers=4,
  drop_last=False
)
```



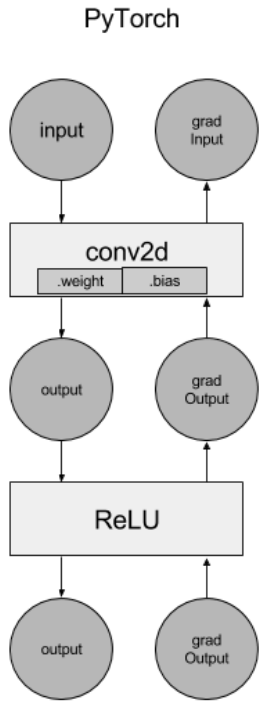
NN PACKAGE

```python
import torch
import torch.nn.functional as F


class MNISTConvNet(torch.nn.Module):

    def __init__(self):

        # call superclass constructor
        super().__init__()

        self.conv1 = torch.nn.Conv2d(1, 10, 5)
        self.pool1 = torch.nn.MaxPool2d(2, 2)
        self.conv2 = torch.nn.Conv2d(10, 20, 5)
        self.pool2 = torch.nn.MaxPool2d(2, 2)
        self.fc1 = torch.nn.Linear(320, 50)
        self.fc2 = torch.nn.Linear(50, 10)


    def forward(self, input):
        x = self.pool1(F.relu(self.conv1(input)))
        x = self.pool2(F.relu(self.conv2(x)))

        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))[]
        x = F.relu(self.fc2(x))

        return x
```



Intermediate states are held in modules

Harder to share weights (manually clone modules and share weights)

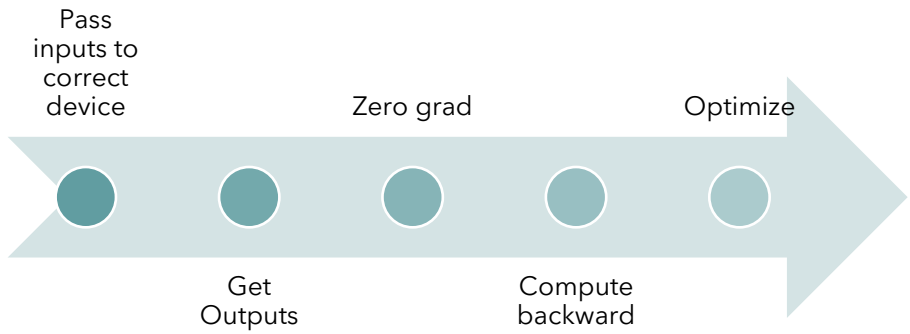


Intermediate states are held in the compute graph

Much easier to share weights (reuse the same module multiple times)

TRAINING LOOP

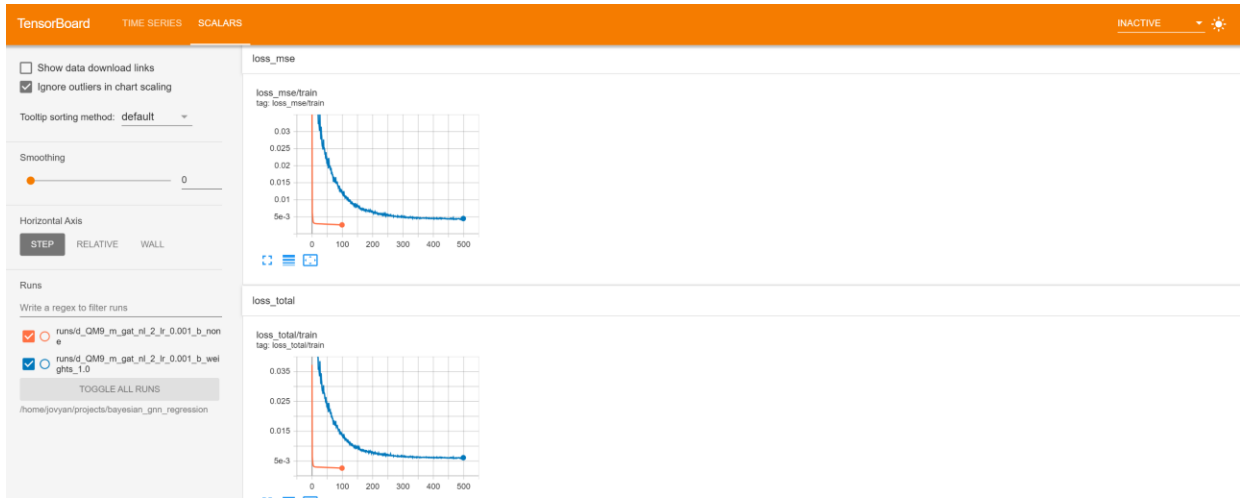
```python
# train mode
model.train()

# initialize vectors
losses = []

# train step loop
for images, labels in train_data:
    # pass images and labels to the correct device
    images = images.to(device)
    labels = labels.to(device)

    # compute outputs and loss
    outputs = model(images)
    loss_value = loss(outputs, labels.long())

    # compute gradient and update parameters
    optimizer.zero_grad()
    loss_value.backward()
    optimizer.step()

    # add metrics to vectors
    losses.append(loss_value.item())

    # progress bar step
    progress_bar.update()

# write results on tensorboard
writer.add_scalar("loss/train", np.mean(losses), epoch)
```



VAL LOOP

```python
# evaluation mode
model.eval()
with torch.no_grad():
    # initialize vector
    losses = []

    # val step loop
    for images, labels in val_data:
        # pass images and labels to the correct device
        images = images.to(device)
        labels = labels.to(device)

        # compute outputs and loss
        outputs = model(images)
        loss_value = loss(outputs, labels.long())

        # add metrics to vectors
        losses.append(loss_value.item())

    # write results on tensorboard
    writer.add_scalar("losses/val", np.mean(losses), epoch)
```

TENSORBOARD




```python
from torch.utils.tensorboard import SummaryWriter

writer: SummaryWriter = SummaryWriter(f"runs/{name}")
writer.add_scalar("train/loss", np.mean(losses), epoch)
```

REFERENCES

- pytorch.org
- philkr.net/cs342
- docs.python.org
- mypy.readthedocs.io
- blog.finxter.com
- valuecoders.com
- https://d2l.ai/
- https://www.dailydoseofds.com/pytorch-models-are-not-deployment-friendly-supercharge-them-with-torchscript/
- https://realpython.com/