A brief, non-conclusive overview of some good coding practices

Follow Style Guidelines

- PEP 8: https://peps.python.org/pep-0008/

- Google Style Guide: https://google.github.io/styleguide/pyguide.html

- If you code using an IDE, you can install linters and code formatters to help you with this. (flake8, black, …)

Docstrings, type hinting, exceptions and comments

```python
def cov(X, rowvar = False):
    X = X.clone()
    if X.dim() < 2:
        X = X.view(1, -1)
    if not rowvar and X.size(0) != 1:
        X = X.t()
    fact = 1.0 / (X.size(1) - 1)
    X -= torch.mean(X, dim=1, keepdim=True)
    Xt = X.t()
    return fact * X.matmul(Xt).squeeze()
```

```python
def cov(X: Tensor, rowvar: bool = False) -> Tensor:
    """Estimate a covariance matrix given data.

    Covariance indicates the level to which two variables vary together.
    If we examine N-dimensional samples, `X = [x_1, x_2, ... x_N]^T`,
    then the covariance matrix element `C_{ij}` is the covariance of
    `x_i` and `x_j`. The element `C_{ii}` is the variance of `x_i`.

    Args:
        X: A 1-D or 2-D array containing multiple variables and observations.
            Each row of `X` represents a variable, and each column a single
            observation of all those variables.
        rowvar: If `rowvar` is True, then each row represents a
            variable, with observations in the columns. Otherwise, the
            relationship is transposed: each column represents a variable,
            while the rows contain observations.

    Returns:
        The covariance matrix of the variables.
    """
    X = X.clone()
    if X.dim() > 2:
        raise ValueError("m has more than 2 dimensions")
    if X.dim() < 2:
        X = X.view(1, -1)
    if not rowvar and X.size(0) != 1:
        X = X.t()
    # m = m.type(torch.double)  # uncomment this line if desired
    fact = 1.0 / (X.size(1) - 1)
    X -= torch.mean(X, dim=1, keepdim=True)
    Xt = X.t()  # if complex: mt = m.t().conj()
    return fact * X.matmul(Xt).squeeze()
```

Vectorization

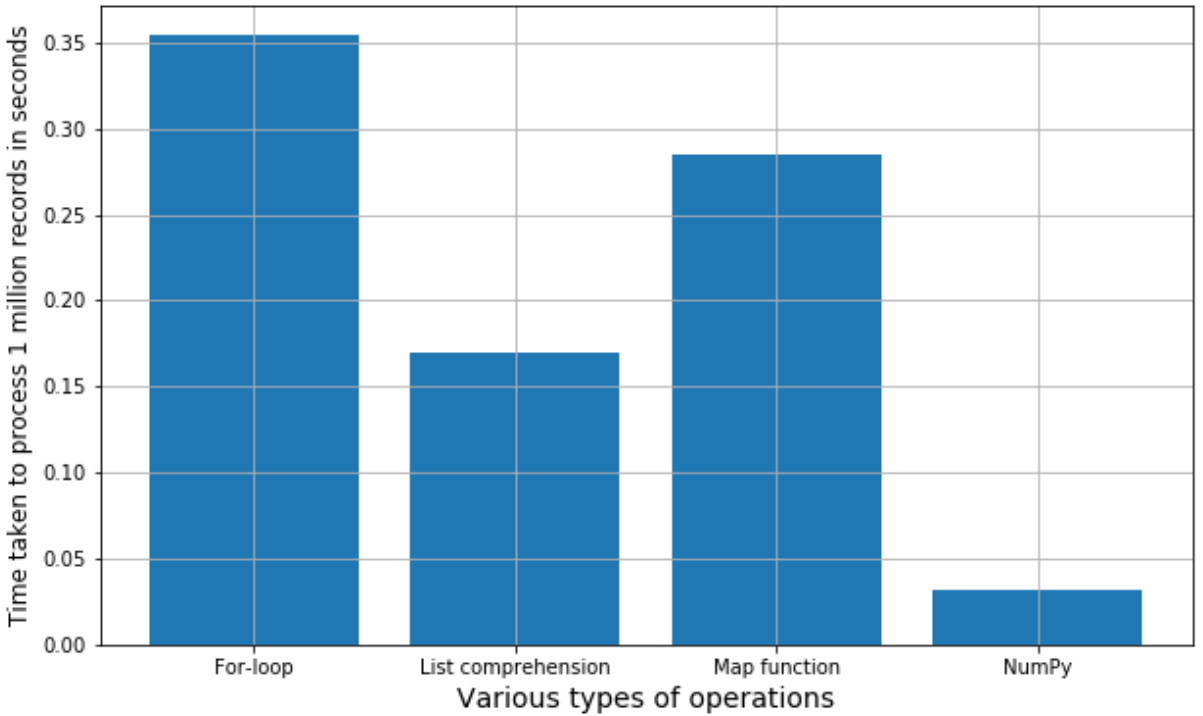
```
l1 = [1e1,1e2,1e3]
l2 = []

t1 = time.time()
for item in l1:
    l2.append(lg10(item))
t2 = time.time()
speed.append(t2-t1)

vs.

v1 = np.array(l1)

t1 = time.time()
v2 = np.log10(v1)
t2 = time.time()
speed.append(t2-t1)
```



From: https://medium.com/productive-data-science/why-you-should-forget-for-loop-for-data-science-code-and-embrace-vectorization-696632622d5f

Vectorization

```
C = A@B
```

vs.

```
C = []
for i in range(len(A)):
    row = []
    for j in range(len(B[0])):
        product = 0
        for v in range(len(A[i])):
            product += A[i][v] * B[v][j]
        row.append(product)
    C.append(row)
```

Loops

Loop over elements rather than index of elements for items in list

```
l = [0,1,2,3]

for i in range(len(l)):
    print(l[i])

vs.

for item in l:
    print(item)

for i, item in enumerate(l):
    print(i, item)
```

Loops

Use zip to iterate over multiple lists

```
l1 = [0,1,2,3]
l2 = [3,2,1,0]

for i in range(len(l1)):
    print(l1[i], l2[i])

vs.

for i, j in zip(l1, l2):
    print(i, j)
```

Loops

Looping over dictionaries

```
d = {"a": 1, "b": 2, "c": 3}

for key in d.keys():
    print(key)

vs.

for key in d:
    print(key)

for key, val in d.items():
    print(key, val)
```

Conditionals

```
If cond == True:
    pass

vs.

if cond:
    pass
```

Strings

```
result = 1.23456789123456789

print("The result is " + str(result) + " units.")


vs.


print("The result is %s units." %result)

print(f"The result is {result} units.")

print("The result is {:.2f} units.".format(result)) # prints only
2 decimals
```

List/Dict comprehensions

```
l = []
for x in range(10):
    l.append(x**2)

vs.

l = [x**2 for x in range(10)]
```

```
l1 = ["0","1","2","3"]
l2 = [0,1,2,3]
d = {}
for key, val in zip(l1,l2):
    d[key] = val

vs.

d = {key: val for key, val in zip(l1, l2)}
```

Context managers

```
f = open("text.txt", "r"):
text = f.read()
f.close()

vs.

with open("text.txt", "r") as f:
    text = f.read()
```

Type checking

```
if type(x) == float:
    pass
```

vs.

```
if isinstance(x, float):
    pass
```

```
if x == None:
    pass
```

vs.

```
if x is None:
    pass
```

Notation

```
x = 100000
y = 0.00001

vs.

x = 1e5
y = 1e-5
```

The Walrus (for Python >= v3.8)

```
while (user_answer := input(f"\n{question} ")) not in valid_answers:
    print(f"Please answer one of {', '.join(valid_answers)}")



[value for num in numbers if (value := slow(num)) > 0]
```

Avoid

`From module import *`

- Overwrites functions that are already within the namespace, i.e. `from numpy import *` will overwrite `sum()` with `numpy.sum()`.

Version control and collaborative coding

Consider working with Git / GitHub

Avoid merge conflicts with jupyter notebooks

- `nbdev` provides git hooks that clean notebooks of unecessary metadata and prevent your notebooks from braking during merge conflicts.

- you can check out how to install it here:

github.com/jnsbck/nbdev_demo.git