



Basics for NLP


Iulia, Rishu, & Chris

WHEN YOU FORGET TO CAPITALYZE THE
BOOL IN PYTHON



Overview of the Lesson

- PEP8 Code Formatting
 - What is it?
 - Why use it?
- PEP8 Guidelines
 - Code Layout
 - String Quotes & Comments
 - Import Statements
 - Function & Class Design
- PEP8 Guidelines
 - Naming Conventions
 - Statements & Expressions
 - Error Handling
 - Best Practices
- PEP8 Libraries
 - Linters
 - Autoformatters

A decorative graphic on the left side of the slide. It features a cluster of colorful paint splashes in shades of blue, green, yellow, and orange. To the right of the splashes are several overlapping geometric shapes, including a dark gray parallelogram and a light gray parallelogram, creating a modern, abstract design.

PEP8 Code Formatting

What is PEP8 Guidelines?

- **Aliases:**
 - PEP8
 - PEP-8
 - Python Enhancement Proposal
- **Authors**
 - Guido van Rossum
 - Barry Warsaw
 - Nick Coghlan
- **What did it aim to do?**
 - “Style Guide for Python Code”
 - Very similar to PEP7 for C
- **How many guidelines are there?**
 - ~30 Rules broken up into 6/8 categories
 - Will we go over all of them...?
 - No, that'd be crazy... unless?



Why use PEP8?

- You probably were already taught a lot of PEP8 when you first were taught Python
 - Readability
 - Consistency
 - Easy Collaboration
 - Maintainability

What the heck is a PEP?

“We intend PEPs to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.”

- Probably Barry Warsaw

So PEP8 is a style guide?

"A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

However, know when to be inconsistent – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

In particular: do not break backwards compatibility just to comply with this PEP!"

- Probably Guido van Rossum

A decorative graphic on the left side of the slide. It features a cluster of colorful paint splashes in shades of blue, green, yellow, and orange. To the right of the splashes are several overlapping geometric shapes, including a dark gray triangle and a light gray parallelogram, creating a modern, abstract design.

PEP8 Guidelines


PEP8 Guidelines Breakdown

What you Probably Know

- Indentation
- Naming Convention
- Whitespace in Statements & Expressions
- Function & Class Design

What you Probably DIDN'T Know

- Code Layout
- Comments
- Best Practices
- Error Handling (Future Class)



What you Probably Know

Indentation

Naming Convention

Whitespace

Function & Class Design



Indentation

- Subcategory of Code Layout
 - Tabs vs Spaces
 - After Line Breaks
 - Closing Brackets

Tabs vs Spaces

- Spaces are the preferred indentation method.
- Tabs should be used solely to remain consistent with code that is already indented with tabs.
- Python disallows mixing tabs and spaces for indentation.

Shell

```
$ python2 -tt code.py
File "code.py", line 3
    print(i, j)
           ^
TabError: inconsistent use of tabs and spaces in indentation
```

Shell

```
$ python2 -t code.py
code.py: inconsistent use of tabs and spaces in indentation
```

After Line Break pt 1

- Line length is generally 79 characters, so you may need to break up a particularly long line
- Most IDEs mark this line on your screen already

Python

```
def function(arg_one, arg_two,  
            arg_three, arg_four):  
    return arg_one
```


After Line Break pt 2

- What about in situations where a nested item is indented upon?
- There are two ways to go about that ->
- There is also something called a Hanging Indent

Python

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

In this case, PEP 8 provides two alternatives to help improve readability:

- Add a comment after the final condition. Due to syntax highlighting in most editors, this will separate the conditions from the nested code:

Python

```
x = 5
if (x > 3 and
    x < 10):
    # Both conditions satisfied
    print(x)
```

- Add extra indentation on the line continuation:

Python

```
x = 5
if (x > 3 and
    x < 10):
    print(x)
```

Group Practice

- Given the Indentation Rules
 - Do the comment method
 - Do the extra indentation method

```
if (this_is_one_thing and  
    that_is_another_thing):  
    do_something()
```

Group Practice

- Given the Indentation Rules
 - Do the comment method
 - Do the extra indentation method

```
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# Add a comment, which will provide some distinction in editors
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
    and that_is_another_thing):
    do_something()
```

Closing Brackets

- Use braces/brackets on new lines to help space out your code

- Line up the closing brace with the first non-whitespace character of the previous line:

Python

```
list_of_numbers = [  
    1, 2, 3,  
    4, 5, 6,  
    7, 8, 9  
]
```

- Line up the closing brace with the first character of the line that starts the construct:

Python

```
list_of_numbers = [  
    1, 2, 3,  
    4, 5, 6,  
    7, 8, 9  
]
```



Naming Conventions

- General Guidelines
- Explicit Names

General Guidelines

- Make sure that your naming convention is consistent for each type that you are using
 - (this is technically like 8/9 guidelines right here)

Type	Naming Convention	Examples
Function	Use a lowercase word or words. Separate words by underscores to improve readability.	<code>function</code> , <code>my_function</code>
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	<code>x</code> , <code>var</code> , <code>my_variable</code>
Class	Start each word with a capital letter. Do not separate words with underscores. This style is called camel case or pascal case .	<code>Model</code> , <code>MyClass</code>
Method	Use a lowercase word or words. Separate words with underscores to improve readability.	<code>class_method</code> , <code>method</code>
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	<code>CONSTANT</code> , <code>MY_CONSTANT</code> , <code>MY_LONG_CONSTANT</code>
Module	Use a short, lowercase word or words. Separate words with underscores to improve readability.	<code>module.py</code> , <code>my_module.py</code>
Package	Use a short, lowercase word or words. Do not separate words with underscores.	<code>package</code> , <code>mypackage</code>

Explicit Names

- Make sure that whatever you label is as unambiguous as possible

Group Practice

- How would you improve these names?

Python

>>>

```
>>> # Not recommended
>>> x = 'John Smith'
>>> y, z = x.split()
>>> print(z, y, sep=', ')
'Smith, John'
```

Explicit Names

- Make sure that whatever you label is as unambiguous as possible

Group Practice

- How would you improve these names?

Python

>>>

```
>>> # Not recommended
>>> x = 'John Smith'
>>> y, z = x.split()
>>> print(z, y, sep=', ')
'Smith, John'
```

Python

>>>

```
>>> # Recommended
>>> name = 'John Smith'
>>> first_name, last_name = name.split()
>>> print(last_name, first_name, sep=', ')
'Smith, John'
```



Whitespace in Statements & Expressions

- Whitespace
 - When to add
 - When to remove

Whitespace: When to Add

- Use it with binary operators

Group Practice

- What happens if there are more than one operator?

- Assignment operators (=, +=, -=, and so forth)
- Comparisons (==, !=, >, <, >=, <=) and (is, is not, in, not in)
- Booleans (and, not, or)

Whitespace: When to Add

- Use it with binary operators

Group Practice

- What happens if there are more than one operator?

- Assignment operators (`=`, `+=`, `-=`, and so forth)
- Comparisons (`==`, `!=`, `>`, `<`, `>=`, `<=`) and (`is`, `is not`, `in`, `not in`)
- Booleans (`and`, `not`, `or`)

Python

```
# Recommended
y = x**2 + 5
z = (x+y) * (x-y)

# Not Recommended
y = x ** 2 + 5
z = (x + y) * (x - y)
```

Whitespace: When to Remove pt 1

- When using if statements with multiple conditions
- Avoid Trailing Whitespace


Python

```
# Recommended
if x>5 and x%2==0:
    print('x is larger than 5 and divisible by 2!')
```

Python

```
# Not recommended
if x > 5 and x % 2 == 0:
    print('x is larger than 5 and divisible by 2!')
```


Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces 
- Before a comma, semicolon, or colon
- Before open parenthesis that starts an argument
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis
- Aligning operators

Python

```
# Recommended  
my_list = [1, 2, 3]  
  
# Not recommended  
my_list = [ 1, 2, 3, ]
```

Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces
- Before a comma, semicolon, or colon ←
- Before open parenthesis that starts an argument
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis
- Aligning operators


Python

```
x = 5  
y = 6
```

```
# Recommended  
print(x, y)
```

```
# Not recommended  
print(x , y)
```

Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces
- Before a comma, semicolon, or colon
- Before open parenthesis that starts an argument 
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis
- Aligning operators

Python

```
def double(x):  
    return x * 2
```

Recommended

```
double(3)
```

Not recommended

```
double (3)
```

Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces
- Before a comma, semicolon, or colon
- Before open parenthesis that starts an argument
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis
- Aligning operators



Python

Recommended

list[3]

Not recommended

list [3]

Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces
- Before a comma, semicolon, or colon
- Before open parenthesis that starts an argument
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis ←
- Aligning operators

Python


Recommended

tuple = (1,)

Not recommended

tuple = (1,)

Whitespace: When to Remove pt 2

- Inside parenthesis, brackets, or braces
- Before a comma, semicolon, or colon
- Before open parenthesis that starts an argument
- Before open brackets that start an index/slice
- Between a trailing comma or closing parenthesis
- Aligning operators 

Python

```
# Recommended  
var1 = 5  
var2 = 6  
some_long_var = 7
```

```
# Not recommended  
var1      = 5  
var2      = 6  
some_long_var = 7
```




Function & Class Design

- Subcategory of Code Layout
 - Function Design
 - Class Design

Class & Function Design

- Use simple classes before complex classes
- Keep functions that are not meant to be global within a class (within reason)

```
class Calculator:
    """A simple calculator class."""

    def __init__(self):
        """Initialize the calculator."""
        self.total = 0

    def add(self, num):
        """Add a number to the total."""
        self.total += num

    def subtract(self, num):
        """Subtract a number from the total."""
        self.total -= num

    def multiply(self, num):
        """Multiply the total by a number."""
        self.total *= num

    def divide(self, num):
        """Divide the total by a number."""
        if num != 0:
            self.total /= num

    def get_total(self):
        """Get the current total."""
        return self.total
```



What you Probably *Don't* Know

Code Layout

Comments

Other Best Practices



Code Layout

- Blank Lines
- Source File Encoding
- Imports
- Module Level Dunder Names

Blank Lines



- Top level Functions get two blank lines
- Method definitions in a class are given one blank line before and after
- Extra blanks may be used to separate groups of related functions
- Use blank lines in functions to break up steps

```
import module1
import module2

CONSTANT_VALUE = 42

def function1():
    # Code for function1
    pass

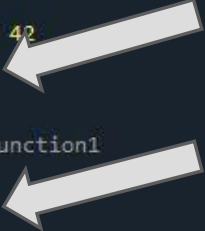
def function2():
    # Code for function2
    pass

class MyClass:
    def __init__(self):
        self.attribute = None

    def method1(self):
        # Code for method1
        pass

    def method2(self):
        # Code for method2
        pass

# Blank line to separate top-level functions and class
def other_function():
    # Code for other_function
    pass
```



Blank Lines

- Top level Functions get two blank lines
- Method definitions in a class are given one blank line before and after
- Extra blanks may be used to separate groups of related functions
- Use blank lines in functions to break up steps

```
import module1
import module2

CONSTANT_VALUE = 42

def function1():
    # Code for function1
    pass

def function2():
    # Code for function2
    pass


class MyClass:
    def __init__(self):
        self.attribute = None

    def method1(self):
        # Code for method1
        pass

    def method2(self):
        # Code for method2
        pass

# Blank line to separate top-level functions and class
def other_function():
    # Code for other_function
    pass
```


Blank Lines

- Top level Functions get two blank lines
- Method definitions in a class are given one blank line before and after
- Extra blanks may be used to separate groups of related functions 
- Use blank lines in functions to break up steps

```
# Blank line to separate top-level functions and class
def other_function():
    # Code for other_function
    pass

def data_preprocessing(data):
    cleaned_data = remove_duplicates(data)
    normalized_data = normalize_data(cleaned_data)
    shuffled_data = shuffle_data(normalized_data)

    return shuffled_data
```

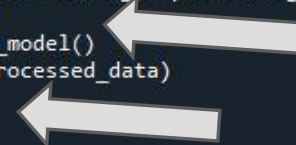


Blank Lines


- Top level Functions get two blank lines
- Method definitions in a class are given one blank line before and after
- Extra blanks may be used to separate groups of related functions
- Use blank lines in functions to break up steps



```
def model_training(data):  
    preprocessed_data = data_preprocessing(data)  
  
    model = create_model()  
    model.fit(preprocessed_data)  
  
    return model
```



Imports

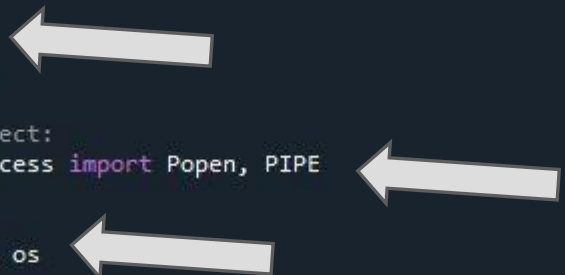
- Imports should usually be on separate lines 
- Imports should be grouped in the following order:
 - Standard library imports.
 - Related third party imports.
 - Local application/library specific imports.
- You should put a blank line between each group of imports.

```
# Correct:
import os
import sys

# Also Correct:
from subprocess import Popen, PIPE

# Wrong:
import sys, os

import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```



Imports

- Imports should usually be on separate lines
- 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.

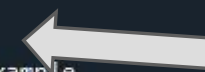


```
# Correct:
import os
import sys

# Also Correct:
from subprocess import Popen, PIPE

# Wrong:
import sys, os

import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```



Imports



- Imports should usually be on separate lines
- 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.

```
# Correct:
import os
import sys

# Also Correct:
from subprocess import Popen, PIPE

# Wrong:
import sys, os

import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```



Module Level Dunder Names

- Module level “dunders” (i.e. names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc. should be placed after the module docstring but before any import statements except from `__future__` imports. Python mandates that future-imports must appear in the module before any other code except docstrings

```
"""This is the example module.  
  
This module does stuff.  
"""  
  
from __future__ import barry_as_FLUFL  
  
__all__ = ['a', 'b', 'c']  
__version__ = '0.1'  
__author__ = 'Cardinal Biggles'  
  
import os  
import sys
```



Comments

- Inline Comments
- Block Comments
- Documentation Strings

Inline Comments

- Use inline comments sparingly.
- Write inline comments on the same line as the statement they refer to.
- Separate inline comments by two or more spaces from the statement.
- Start inline comments with a `#` and a single space, like block comments.
- Don't use them to explain the obvious.

Python

```
x = 5 # This is an inline comment
```

Group Practice

Is this a good inline comment?

Python

```
x = 'John Smith' # Student Name
```


Inline Comments

- Use inline comments sparingly.
- Write inline comments on the same line as the statement they refer to.
- Separate inline comments by two or more spaces from the statement.
- Start inline comments with a `#` and a single space, like block comments.
- Don't use them to explain the obvious.

Python

```
student_name = 'John Smith'
```

Group Practice

Is this a good inline comment?

Python

```
empty_list = [] # Initialize empty list
```

```
x = 5
```

```
x = x * 5 # Multiply x by 5
```

Block Comments

- Indent block comments to the same level as the code they describe.
- Start each line with a `#` followed by a single space.
- Separate paragraphs by a line containing a single `#`.

Python

```
for i in range(0, 10):  
    # Loop over i ten times and print out the value of i, followed by a  
    # new line character  
    print(i, '\n')
```

Python

```
def quadratic(a, b, c, x):  
    # Calculate the solution to a quadratic equation using the quadratic  
    # formula.  
    #  
    # There are always two solutions to a quadratic equation, x_1 and x_2.  
    x_1 = (- b + (b**2 - 4*a*c)**(1/2)) / (2*a)  
    x_2 = (- b - (b**2 - 4*a*c)**(1/2)) / (2*a)  
    return x_1, x_2
```

Documentation Strings

- Surround docstrings with three double quotes on either side, as in `"""This is a docstring"""`.
- Write them for all public modules, functions, classes, and methods.
- Put the `"""` that ends a multiline docstring on a line by itself
 - If it's a one line docstring, keep it all on one line

Python

```
def quadratic(a, b, c, x):  
    """Solve quadratic equation via the quadratic formula.  
  
    A quadratic equation has the following form:  
     $ax^2 + bx + c = 0$   
  
    There always two solutions to a quadratic equation:  $x_1$  &  $x_2$ .  
    """  
    x_1 = (- b + (b**2 - 4*a*c)**(1/2)) / (2*a)  
    x_2 = (- b - (b**2 - 4*a*c)**(1/2)) / (2*a)  
  
    return x_1, x_2
```



Other Best Practices

- String Quotes
- Operator Placement
- Source File Encoding

String Quotes & Source File Encoding

- If you use ` then continue to use them
- If you use " then continue to use them
- BUT DO NOT MIX
 - Why?
- Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

Operator Placement

- Make sure when using mathematical operators to put them at the start of indentation if you make one



```
income = (gross_wages +  
          taxable_interest +  
          (dividends - qualified_dividends) -  
          ira_deduction -  
          student_loan_interest)
```

```
income = (gross_wages  
          + taxable_interest  
          + (dividends - qualified_dividends)  
          - ira_deduction  
          - student_loan_interest)
```



Let's try some examples

- 9 Questions
 - Multiple Answer
 - One is a Trick Question


```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def my_function(arg1, arg2):  
    return arg1 + arg2
```

Naming Convention & whitespace

```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def my_function(arg1, arg2):  
    return arg1 + arg2
```

```
import numpy  
import pandas  
import os
```

Import

```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def my_function(arg1, arg2):  
    return arg1 + arg2
```

```
import numpy  
import pandas  
import os
```

```
x = 10  
y = 20  
z = 30
```

Whitespace around Operators

```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def my_function(arg1, arg2):  
    return arg1 + arg2
```

```
import numpy  
import pandas  
import os
```

```
x = 10  
y = 20  
z = 30
```

```
def my_function():  
    # Some code here  
    return result
```

Indentation

```
def myfunction( arg1,arg2):  
    return arg1+arg2
```

```
import numpy, pandas, os
```

```
x=10  
y=20  
z=30
```

```
def my_function():  
    # Some code here  
    return result
```

```
class myclass:  
    def __init__(self):  
        self.attribute = 10  
    def method_one(self):  
        pass
```

```
def my_function(arg1, arg2):  
    return arg1 + arg2
```

Naming Convention & whitespace

```
import numpy  
import pandas  
import os
```

Import

```
x = 10  
y = 20  
z = 30
```

Whitespace around Operators

```
def my_function():  
    # Some code here  
    return result
```

Indentation

```
class MyClass:  
    def __init__(self):  
        self.attribute = 10  
  
    def method_one(self):  
        pass
```

Naming Convention & Blank Space

```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

```
# This function calculates the result
def calculateResult(arg1, arg2):
    return arg1 + arg2
```



```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

```
# This function calculates the result
def calculateResult(arg1, arg2):
    return arg1 + arg2
```

```
# This function performs a calculation
def calculate_result(arg1, arg2):
    return arg1 * arg2
```

Naming Convention & Whitespace

```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

```
# This function calculates the result
def calculateResult(arg1, arg2):
    return arg1 + arg2
```

```
# This function performs a calculation
def calculate_result(arg1, arg2):
    return arg1 * arg2
```

```
if x > 10 and y < 20:
    print("Condition met")
```

Whitespace

```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

```
# This function calculates the result
def calculateResult(arg1, arg2):
    return arg1 + arg2
```

```
# This function performs a calculation
def calculate_result(arg1, arg2):
    return arg1 * arg2
```

```
if x > 10 and y < 20:
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

Tab vs Space ;D

```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass

    def method_two(self):
        pass
```

```
# This function calculates the result
def calculateResult(arg1, arg2):
    return arg1 + arg2
```

```
# This function performs a calculation
def calculate_result(arg1, arg2):
    return arg1 * arg2
```

```
if x > 10 and y < 20:
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass

    def method_two(self):
        pass
```

Blank Space

```
#This function performs a calculation
def calculateResult(arg1,arg2):
    return arg1 * arg2
```

```
if(x>10 and y<20):
    print("Condition met")
```

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

```
# This function performs a calculation
def calculate_result(arg1, arg2):
    return arg1 * arg2
```

Naming Convention & Whitespace

```
if x > 10 and y < 20:
    print("Condition met")
```

Whitespace

```
def my_function():
    # Step 1
    do_something()

    # Step 2
    do_something_else()
```

Tab vs Space ;D

```
class MyClass:
    def __init__(self):
        self.attribute = 10

    def method_one(self):
        pass
    def method_two(self):
        pass
```

Blank Space



PEP8 Libraries

Linters

Autoformatters

What are PEP8 Libraries?

Linters

- Linters are programs that analyze code and flag errors
- They provide suggestions on how to fix the error.

Example:

- [Pycodestyle](#)
- [Flake8](#)

(these are even found in Atom, Sublime Text, Visual Studio Code, and VIM as extensions)

Autoformatters

- Autoformatters are programs that refactor your code to conform with PEP 8 automatically.

Example:

- [Black](#)
- [Autopep8](#)
- [Yapf](#)