Pythonicness & Packaging

The Zen of Python

Writing programs that actually do what they are supposed to do is just one component of being a good Python programmer.
It's also important to write clean code that is easily understood, even weeks after you've written it.

One way of doing this is to follow the **Zen of Python**, a somewhat tongue-in-cheek set of principles that serves as a guide to programming the Pythoneer way. Use the following code to access the Zen of Python.

```
import this
```

Try It Yourself

Result:
The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Tap **Try It Yourself** to play around with the code!

The Zen of Python

Some lines in the Zen of Python may need more explanation.
Explicit is better than implicit: It is best to spell out exactly what your code is doing. This is why adding a numeric string to an integer requires explicit conversion, rather than having it happen behind the scenes, as it does in other languages.
Flat is better than nested: Heavily nested structures (lists of lists, of lists, and on and on...) should be avoided.
Errors should never pass silently: In general, when an error occurs, you should output some sort of error message, rather than ignoring it.

There are 20 principles in the Zen of Python, but only 19 lines of text.
The 20th principle is a matter of opinion, but our interpretation is that the blank line means "Use whitespace".

The line "There should be one - and preferably only one - obvious way to do it" references and contradicts the Perl language philosophy that there should be more than one way to do it.

PEP

Python Enhancement Proposals (PEP) are suggestions for improvements to the language, made by experienced Python developers.
PEP 8 is a style guide on the subject of writing readable code. It contains a number of guidelines in reference to variable names, which are summarized here:
- modules should have short, all-lowercase names;
- class names should be in the CapWords style;
- most variables and function names should be lowercase_with_underscores;
- constants (variables that never change value) should be CAPS_WITH_UNDERSCORES;
- names that would clash with Python keywords (such as 'class' or 'if') should have a trailing underscore.

PEP 8 also recommends using spaces around operators and after commas to increase readability.

However, whitespace should not be overused. For instance, avoid having any space directly inside any type of brackets.

PEP 8

Other **PEP 8** suggestions include the following:
- lines shouldn't be longer than 80 characters;
- 'from module import *' should be avoided;
- there should only be one statement per line.

It also suggests that you use spaces, rather than tabs, to indent. However, to some extent, this is a matter of personal preference. If you use spaces, only use 4 per line. It's more important to choose one and stick to it.

The most important advice in the PEP is to ignore it when it makes sense to do so. Don't bother with following PEP suggestions when it would cause your code to be less readable; inconsistent with the surrounding code; or not backwards compatible.
However, by and large, following PEP 8 will greatly enhance the quality of your code.

Some other notable PEPs that cover code style:
PEP 20: The Zen of Python
PEP 257: Style Conventions for Docstrings

Function Arguments

Python allows to have function with **varying number of arguments**.
Using ***args** as a function parameter enables you to pass an arbitrary number of arguments to that function. The arguments are then accessible as the tuple **args** in the body of the function.
Example:

```
def function(named_arg, *args):
  print(named_arg)
  print(args)

function(1, 2, 3, 4, 5)
```

Try It Yourself

Result:

```
>>>
1
(2, 3, 4, 5)
>>>
```

The parameter ***args** must come after the named parameters to a function.
The name **args** is just a convention; you can choose to use another.

Default Values

Named parameters to a function can be made optional by giving them a **default value**.
These must come after named parameters without a default value.
Example:

```
def function(x, y, food="spam"):
  print(food)

function(1, 2)
function(3, 4, "egg")
```

Try It Yourself

Result:

```
>>>
spam
egg
>>>
```

In case the argument is passed in, the default value is ignored.
If the argument is not passed in, the default value is used.

Function Arguments

****kwargs** (standing for keyword arguments) allows you to handle named arguments that you have not defined in advance.
The keyword arguments return a dictionary in which the keys are the argument names, and the values are the argument values.
Example:

```
def my_func(x, y=7, *args, **kwargs):
  print(kwargs)

my_func(2, 3, 4, 5, 6, a=7, b=8)
```

Try It Yourself

Result:

```
>>>
{'a': 7, 'b': 8}
>>>
```

a and **b** are the names of the arguments that we passed to the function call.

> The arguments returned by ****kwargs** are not included in ***args**.

Tuple Unpacking

Tuple unpacking allows you to assign each item in an iterable (often a tuple) to a variable.
Example:

```
numbers = (1, 2, 3)
a, b, c = numbers
print(a)
print(b)
print(c)
```

Try It Yourself

Result:

```
>>>
1
2
3
>>>
```

> This can be also used to swap variables by doing **a, b = b, a** , since **b, a** on the right hand side forms the tuple **(b, a)** which is then unpacked.

Tuple Unpacking

A variable that is prefaced with an asterisk (*) takes all values from the iterable that are left over from the other variables.
Example:

```
a, b, *c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
print(b)
print(c)
print(d)
```

Try It Yourself

Result:

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

> Tap **Try It Yourself** to play around with the code!

Ternary Operator

Conditional expressions provide the functionality of if statements while using less code. They shouldn't be overused, as they can easily reduce readability, but they are often useful when assigning variables.
Conditional expressions are also known as applications of the **ternary operator**.
Example:

```
a = 7
b = 1 if a >= 5 else 42
print(b)
```

Try It Yourself

Result:

```
>>>
1
>>>
```

The ternary operator checks the condition and returns the corresponding value.
In the example above, as the condition is true, **b** is assigned 1. If **a** was less than 5, it would have been assigned 42.
Another example:

```
status  = 1
msg = "Logout" if status == 1 else "Login"
```

Try It Yourself

The **ternary** operator is so called because, unlike most operators, it takes **three** arguments.

else

The **else** statement is most commonly used along with the **if** statement, but it can also follow a **for** or **while** loop, which gives it a different meaning.
With the **for** or **while** loop, the code within it is called if the loop finishes normally (when a break statement does not cause an exit from the loop).

Example:

```
for i in range(10):
  if i == 999:
    break
else:
  print("Unbroken 1")

for i in range(10):
  if i == 5:
    break
else:
  print("Unbroken 2")
```

Try It Yourself

Result:

```
>>>
Unbroken 1
>>>
```

The first **for** loop executes normally, resulting in the printing of "Unbroken 1".
The second loop exits due to a **break**, which is why it's **else** statement is not executed.

---

else

The **else** statement can also be used with **try/except** statements.
In this case, the code within it is only executed if no error occurs in the **try** statement.
Example:

```
try:
  print(1)
except ZeroDivisionError:
  print(2)
else:
  print(3)

try:
  print(1/0)
except ZeroDivisionError:
  print(4)
else:
  print(5)
```

Try It Yourself

Result:

```
>>>
1
3
4
>>>
```

Tap **Try It Yourself** to play around with the code!

__main__

Most Python code is either a module to be imported, or a script that does something.
However, sometimes it is useful to make a file that can be both imported as a module and run as a script.
To do this, place script code inside **if __name__ == "__main__"**.
This ensures that it won't be run if the file is imported.
Example:

```
def function():
  print("This is a module function")

if __name__=="__main__":
  print("This is a script")
```

Try It Yourself

Result:

```
>>>
This is a script
>>>
```

When the Python interpreter reads a source file, it executes all of the code it finds in the file. Before executing the code, it defines a few special variables.
For example, if the Python interpreter is running that module (the source file) as the main program, it sets the special **__name__** variable to have a value **"__main__"**. If this file is being imported from another module, **__name__** will be set to the module's name.

__main__

If we save the code from our previous example as a file called **sololearn.py**, we can then import it to another script as a module, using the name **sololearn**.
sololearn.py

```
def function():
  print("This is a module function")

if __name__=="__main__":
  print("This is a script")
```

Try It Yourself

some_script.py

```
import sololearn

sololearn.function()
```

Result:

```
>>>
This is a module function
>>>
```

Basically, we've created a custom module called **sololearn**, and then used it in another script.

Major 3rd-Party Libraries

The Python standard library alone contains extensive functionality.
However, some tasks require the use of third-party libraries. Some major third-party libraries:
Django: The most frequently used web framework written in Python, Django powers websites that include Instagram and Disqus. It has many useful features, and whatever features it lacks are covered by extension packages.
CherryPy and **Flask** are also popular web frameworks.

For scraping data from websites, the library **BeautifulSoup** is very useful, and leads to better results than building your own scraper with regular expressions.

> While Python does offer modules for programmatically accessing websites, such as urllib, they are quite cumbersome to use. Third-party library requests make it much easier to use HTTP requests.

Major 3rd-Party Libraries

A number of third-party modules are available that make it much easier to carry out scientific and mathematical computing with Python.
The module **matplotlib** allows you to create graphs based on data in Python.
The module **NumPy** allows for the use of multidimensional arrays that are much faster than the native Python solution of nested lists. It also contains functions to perform mathematical operations such as matrix transformations on the arrays.
The library **SciPy** contains numerous extensions to the functionality of **NumPy**.

Python can also be used for **game development**.
Usually, it is used as a scripting language for games written in other languages, but it can be used to make games by itself.

> For 3D games, the library **Panda3D** can be used. For 2D games, you can use **pygame**.

Packaging

In Python, the term **packaging** refers to putting modules you have written in a standard format, so that other programmers can install and use them with ease.
This involves use of the modules **setuptools** and **distutils**.
The first step in packaging is to organize existing files correctly. Place all of the files you want to put in a library in the same parent directory. This directory should also contain a file called **__init__.py**, which can be blank but must be present in the directory.
This directory goes into another directory containing the readme and license, as well as an important file called **setup.py**.
Example directory structure:

```
SoloLearn/
  LICENSE.txt
  README.txt
  setup.py
  sololearn/
    __init__.py
    sololearn.py
    sololearn2.py
```

> You can place as many script files in the directory as you need.

Packaging

The next step in packaging is to write the **setup.py** file.
This contains information necessary to assemble the package so it can be uploaded to PyPI and installed with **pip** (name, version, etc.).
Example of a **setup.py** file:

```
from distutils.core import setup
```

```
setup(
  name='SoloLearn',
  version='0.1dev',
  packages=['sololearn',],
  license='MIT',
  long_description=open('README.txt').read(),
)
```

After creating the **setup.py** file, upload it to PyPI, or use the command line to create a binary distribution (an executable installer).
To build a source distribution, use the command line to navigate to the directory containing setup.py, and run the command **python setup.py sdist**.
Run **python setup.py bdist** or, for Windows, **python setup.py bdist_wininst** to build a binary distribution.
Use **python setup.py register**, followed by **python setup.py sdist upload** to upload a package.

Finally, install a package with **python setup.py install**.

Packaging

The previous lesson covered packaging modules for use by other Python programmers. However, many computer users who are not programmers do not have Python installed. Therefore, it is useful to package scripts as executable files for the relevant platform, such as the Windows or Mac operating systems. This is not necessary for Linux, as most Linux users do have Python installed, and are able to run scripts as they are.

For Windows, many tools are available for converting scripts to executables. For example, **py2exe**, can be used to package a Python script, along with the libraries it requires, into a single executable.
PyInstaller and **cx_Freeze** serve the same purpose.

For Macs, use **py2app**, **PyInstaller** or **cx_Freeze**.

End.