Discussion Section # 1 : Basics of Python

Mitchell Valdes

Introduction

Writing good code

- **Goal:** Enable meaningful data manipulation, not become coding experts.
- **Definition:** Good code is easily understandable by humans.
- **Python's Strength:** Inherent readability.

Specifics

- **Clarity:** Code should be clear and concise.
- **Readability:** Easy to follow logic and structure.
- **Simplicity:** Avoid unnecessary complexity.
- **Consistency:** Maintain a uniform coding style.
- **Comments:** Use comments for clarity when needed.
- **Modularity:** Divide code into reusable modules.
- **Error Handling:** Include error-handling mechanisms.
- **Remember:** Our aim is to leverage Python for data tasks effectively.

Python Zen

```
import this
```

```
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!
```

Style Guide

Python Style Guide: PEP-8

- Developed by Python's core developers.
- Aims to enhance code readability.
- Encourages consistent coding practices.

. . .

Our Emphasis:

- **PEP-8 Awareness:** Be familiar with the guidelines.
- **Prioritize Readability:** Focus on making code easy to read.
- **Don't Obsess Over Style:** Don't stress excessively.
- **Key Themes:** Emphasize important style principles.

More on Style

1. Limit the length of a line of code to 79 characters. Having to scroll right to read a line of code is distracting. Some text editors can display a line down the screen to mark the 'edge.'
2. Use blank lines to group related blocks of code.
3. Put a space before and after assignment `=` and comparisons `==`, `!=`, `<`, `<`.

4. Put a space after a comma.

More on Style

Example: Bad Style

```
# Example of point 3.
x=1+5

if x<10:print(x)

# Example of point 4 (and point 3).
print('The value of x is',x)

x_list = [1,3,6,9]

# Example of point 1.
"This is a very long string that should be wrapped across multiple lines. In its current f
```

Example: Good Style

```
# Examples of point 3.
x = 1 + 5

if x < 10:
    print(x)

# Examples of point 4 (and point 3).
print('The value of x is', x)

x_list = [1, 3, 6, 9]

# Example of point 1.
"""
This is a very long string that we are
going to wrap across multiple lines.
In its current form, it's not too difficult to read.
"""
```

Note that both of these code blocks are valid Python code. The second block is just easier to read.

Comments

1. Use comments when needed. Do not comment on the trivial.
2. Comments should be complete sentences. Capitalize the first letter. Use a period at the end.
3. Use a comment on the line before code rather than to the right of the code.
4. Use docstrings whenever you write a function. This rule will make more sense once we have introduced functions next week.

Example:

```
# Examples of point 1.

# Add 1 to x.   <-- No! This is obvious.
x = x + 1

# Examples of point 3.
# This is a comment on the line before the code.
x += 1  # This is a comment to the right of the code.
```

Naming conventions

Different people, languages, and businesses use different naming conventions. The key here is to be consistent.

1. I use all lower case letters for variables and functions and I use underscores _ to make the names more readable.
2. Use all capital letters to denote a constant value.
3. Try to make function and variable names self-documenting.

Naming conventions

Example: Bad Style

```
x = "Alice"
y = 46
z = 23.32

print(f"{x} income is ${y} while US GDP is ${z} trillion")
```

Example: Good Style

```
name = "Alice"
income = 46

USGDP = 23.32

print(f"{name} income is ${income} while US GDP is ${USGDP} trillion")
```

Practice: Writing good code

Fix up the code below.

```
#everyone's hours
Hrsworked_Bob =37
Hrsworked_Alice = 42
hrswored_Clarice=45.6


wage = 23.5000   # The wage.

x = Hrsworked_Bob* wage +Hrsworked_Alice*wage+wage*hrswored_Clarice

print(    'The total wage bill is',x)
```

```
The total wage bill is 2928.1000000000004
```

String formatting

We work with text a lot:

- We print words and numbers to the screen and to files.
- We add text to figures to label axes and call out interesting observations.
- We analyze text as data.

String formatting

- Next we will learn the basics of string formatting (or string interpolation).
- String formatting is a way to insert values into a string.
- String formatting can get very sophisticated, but we need only a few simple things. As usual, you can always Google for 'python string format' for more.

Simplest way: Strign concatenation

- In python, we can concatenate strings using the `+` operator.
- We can also concatenate strings with other data types, but we need to convert the other data type to a string first.

Example: String concatenation

This code will produce an error. Why?

```
name = "Pedro"
age = "23"
print("His name is " + name + " and he is " + age + " years old.")
```

```
His name is Pedro and he is 23 years old.
```

```
income = 100000

print("His name is " + name + " and he makes $" + income + " per year.")
```

```
can only concatenate str (not "int") to str
```

How can we fix it?

. . .

```
print("His name is " + name + " and he makes $" + str(income) + " per year.")
```

```
His name is Pedro and he makes $100000 per year.
```

The `str()` function converts the `income` variable to a string.

Ways to format strings

There are three ways in which Python can format strings for us:

- Old way: `%` operator

```
text = "The value of x is %d" % x
```

- New way: `.format()` method

```
text = "The value of x is {value}".format(value = x)
```

- Newest way: `f-strings`

```
text = f"The value of x is {x}"
```

Note that in all three cases Python handles the conversion of `x` to a string for us.

Ways to format strings

```
# the following all produce the same output
food = 'apple'
name = 'Bob'

# old method
print('%s had a(n) %s for breakfast' % (name, food))
```

```
Bob had a(n) apple for breakfast
```

```
# .format() method
print('{person} had a(n) {item} for breakfast'.format(person = name, item = food))
```

```
Bob had a(n) apple for breakfast
```

```
# "f-string" method
print(f'{name} had a(n) {food} for breakfast')
```

```
Bob had a(n) apple for breakfast
```

I prefer the `f-string` method. It is the newest and easiest to read.

```
#| echo: true
# the following all produce the same output
food = 'apple'
name = 'Bob'

# old method
print('%s had a(n) %s for breakfast' % (name, food))

# .format() method
print('{person} had a(n) {item} for breakfast'.format(
person = name, item = food))

# "f-string" method
print(f'{name} had a(n) {food} for breakfast')
```

Formatting floats

- We display a decimal value as percentage using the `x.yf` format specifier. Where `x` is the total number of digits and `y` is the number of digits after the decimal point.

```
foo = 1/3
# This is default print
print(f'foo = {foo}')
```

```
foo = 0.3333333333333333
```

```
# This is the default float format.
print(f'foo = {foo:f}')
```

```
foo = 0.333333
```

```
# This is a formatted float with 3 digits after the decimal point.
print(f'foo = {foo:5.3f}')
```

```
foo = 0.333
```

```
# Now the space reserved for the number is 8
print(f'foo = {foo:8.3f}'.format(foo))
```

```
foo =    0.333
```

```
# The zero before the 8 prints out the leading zeros.
print(f'foo = {foo:08.3f}')
```

```
foo = 0000.333
```

```
# We can use commas for easier reading.
big_float = 5692348925.2
print(f'An easy to read  big float is = {big_float:16,.2f}')
```

```
An easy to read  big float is = 5,692,348,925.20
```

Practice: String formatting

1. What happens if the number is larger than the width we specify? Try printing

```
# Question 1
big_float = 123456789123456.7899
```

to the screen with the format `5.2f`. What happened?

. . .

```
print(f"big_float = {big_float:5,.2f}")
```

```
big_float = 123,456,789,123,456.80
```

Practice: String formatting

2. We often express numbers in percentage terms. Experiment with the `{position:4.2%}` format code (where the 4 and 2 are arbitrary numbers). Print each of the following to the screen:
    1. `x = 0.03` as a percentage with 1 decimal place
    2. `y = 0.65297` as a percentage with 2 decimal places
    3. `z = 0.00056` as a percentage with 3 decimal places

. . .

```
x = 0.03
print(f"x = {x:4.2%}")
```

```
x = 3.00%
```

```
y = 0.65297
print(f"y = {y:4.2%}")
```

```
y = 65.30%
```

```
z = 0.00056
print(f"z = {z:4.2%}")
```

```
z = 0.06%
```

Practice: String formatting

3. Suppose that we wanted our x, y, and z to line up on the decimal place.

03.000% 65.30% 00.056%

Try out the `{value:06.2%}` code to add leading zeros.

. . .

```
print(f"x = {x:06.2%}")
```

```
x = 03.00%
```

```
print(f"y = {y:06.2%}")
```

```
y = 65.30%
```

```
print(f"z = {z:06.2%}")
```

```
z = 00.06%
```

. . . or

```
print(f"x = {x:06.2%}\ny = {y:06.2%}\nz = {z:06.2%}")
```

```
x = 03.00%
y = 65.30%
z = 00.06%
```