COMP10001 Foundations of Computing
PEP8 and Commenting; Debugging

Semester 1, 2019
Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— version: 1514, date: April 1, 2019 —

Lecture Agenda

- This lecture:
  - PEP8
  - Commenting
  - Debugging

Lecture Outline

Going Pythonic

- As you are perhaps picking up on gradually, Python is big on stylistics and readability, and the idea that there is one "right" way of doing things
- In this vein, Python has stylistic guidelines on "right" and "wrong" ways of writing code (= PEP8), some of which we take on board in this subject, and start automatically checking for in your code from Worksheet 10 (and for all the projects)

Going Pythonic: Indentation

- Indentation should always be in multiples of 4 spaces (which Grok does by default; so long as you don't modify this/code outside Grok, you should be fine):
- There should be a single space between operators and their operands, and after commas:

✗ Wrong:

```
w_len=0
for i in ('a','b'):
  w_len=w_len+1
```

✓ Right:

```
w_len = 0
for i in ('a', 'b'):
    w_len = w_len + 1
```


Going Pythonic: Whitespace in Expressions and Statements

- Operators should have a single space either side of them:

✗ Wrong:

```
a=1
b      = 2
```

✓ Right:

```
a = 1
b = 2
```

Going Pythonic: Whitespace in Expressions and Statements

- Single space after commas and `dict` colons, no spaces before; no spaces around brackets or between function and arguments

✗ Wrong:

```
a = ( 1 , 2 )
b = { 'a' : 3}
c = 'string'[2: ]
d = len (c)
```

✓ Right:

```
a = (1, 2)
b = {'a': 3}
c = 'string'[2:]
d = len(c)
```


Going Pythonic: Avoid Long Lines

- Lines must not exceed 79 characters

✗ Wrong:

```
def fun(thing):
    '''take `thing` and do nothing to it, but document it in a l
    if 0 > 1 and "totoro" in "avengers" and "abracadabra".isalph
        pass
    return thing
```


Going Pythonic: Avoid Long Lines

METHOD 1 (preferred): using unclosed parentheses, and match operators with operands:

✓ Right:

```
def fun(thing):
    '''take `thing` and do nothing to it, but
    document it in a long-winded way'''
    if (0 > 1 and "totoro" in "avengers"
            and "abracadabra".isalpha()):
        pass
    return thing
```


Going Pythonic: Avoid Long Lines

- METHOD 2 (deprecated): using \ to indicate that the line continues onto the following line:

✓ Right:

```
def fun(thing):
    '''take `thing` and do nothing to it, but
    document it in a long-winded way'''
    if 0 > 1 and "totoro" in "avengers" \
            and "abracadabra".isalpha():
        pass
    return thing
```


Going Pythonic: Segmentation

- Use blank lines to separate logical sections:

✗ Wrong:

```
def w_count(word):
    w_len = 0
    for char in word:
        w_len += 1
    return w_len
```

✓ Right:

```
def w_count(word):

    # letter count
    w_len = 0

    # count the letters
    for char in word:
        w_len += 1

    return w_len
```


Going Pythonic: Don't Stack

- Avoid multiple statements on the same line
- Always start a new line after `if`, `elif`, `else`, `while`, `for`, etc.

✗ Wrong:

```
a = True; b = 0
if a: b += 1
else: b += 2
```

✓ Right:

```
a = True
b = 0
if a:
    b += 1
else:
    b += 2
```

Going Pythonic: Comment Sensibly

- Make sure your comments do not contradict your code
- Do not state the obvious in comments

✗ Wrong:

```
# initialise `a` to 0
a = 0

# decrement `a`
a += 1
```

✓ Right:

```
# count of letters
a = 0

...

a += 1
```


Going Pythonic: Name Sensibly

- Never use the characters `l`, `O` or `I` as single-character variable names
- Use self-descriptive variable names

✗ Wrong:

```
l = I = 1
O = 0
```

✓ Right:

```
count = min_val = 1
max_val = 0
```


Going Pythonic: Function Names

- Function names should be lowercase, with words separated by underscores as necessary to improve readability

✗ Wrong:

```
def DOSOMETHING(x):
    ...
```

✓ Right:

```
def do_something(x):
    ...
```


Going Pythonic: Constant Names

- Constants should be written in all capital letters with underscores separating words, and listed in the "header" of your code

✗ Wrong:

```
k = 3

def fun(i, k=k):
    ....
```

✓ Right:

```
CHAR_SIZE = 3

def fun(i, k=CHAR_SIZE):
    ....
```


Going Pythonic: Comparing Booleans

- Don't compare Boolean values to `True` or `False` using ==

✗ Wrong:

```
if val == False:
    ....
elif val == True:
    ....
```

```
if val2 == True:
    return True
else:
    return False
```

✓ Right:

```
if not val:
    ....
else:
    ....
```

```
return val2
```


Lecture Outline

Class Exercise: Debug the Following

```
def substrn(sup, sub)
    sub_len = len(Sub)
    for i in range(len(sup) - sub_len):
        if sup[i:i + sub_len] == sub:
            n += 1
    print("n")
```


Comments

- Job much easier if there was a description of what the function should do:

```
def substrn(sup, sub)
    '''
    Calculate the number of times `sub`
    occurs in `sup`
    '''
    sub_len = len(Sub)
    for i in range(0, len(sup)-sub_len+1):
        if sup[i:i+sub_len] == sub:
            n += 1
    print("n")
```


Comments

- Job easier again if we knew what each chunk of code was supposed to do:

```
def substrn(sup, sub)
    '''
    Calculate the number of times `sub`
    occurs in `sup`
    '''
    # pre-calculate length of `sub`
    sub_len = len(Sub)

    # generate all substrings of `sup` of length
    # `sub_len`, and test for identity with `sub`
    for i in range(0, len(sup)-sub_len+1):
        if sup[i:i+sub_len] == sub:
            n += 1

    print("n")
```


Functions and Docstring-style Commenting

- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

```
def Celcius2Fahrenheit(n):
    """Calculate the (float)
    Fahrenheit equivalent
    of a temperature in Celcius"""
    return 9.0*n/5 + 32
```


Functions and Docstring-style Commenting

- It is possible to access the `__doc__` for a function via `help`, e.g. given:

```
def seconds_in_year(days=365):
    """Calculate seconds in a year"""
    return days*24*60*60
```

```
>>> help(seconds_in_year)
Help on function seconds_in_year in module __main__:

seconds_in_year(days=365)
    Calculate seconds in a year
```


Comments

✗ Wrong:

```
def f(x):
    '''This is a function of parameter x
    that returns the length of x
    squared.'''
    return len(x)**2
```

Don't describe Python syntax; the reader knows Python:

✓ Right:

```
def grid_cell_size(x):
    '''Calculate the size of a square grid cell
    of side length `x`'''
    return len(x)**2
```

Comments

✗ Wrong:

```
c = 0                  # a variable to count
for i in word:         # a loop
    if i in 'aeiou': # is i in aeiou
        c += 1         # add one to count
```

Succinct description of each logic logic for a block in English.
Meaningful variable names help readability.

Comments

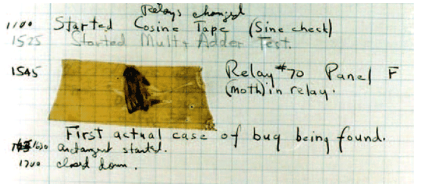
✓ Right:

```
# use `count` to calculate number of vowels
# in `word`
count = 0
for character in word:
    if character in 'aeiou':
        count += 1
```


Lecture Outline

Bugs

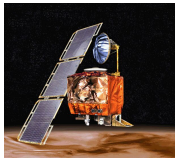
- A (software) "bug" is an error/flaw in a piece of code that leads to a malfunction
- The first attested computer "bug" (Grace Hopper, Harvard Mark II):



- So what's the big deal?

A Bug in Action: Mars Climate Orbiter

- **Ideal:** establish an orbit around Mars, and study the weather, climate, etc of Mars in tandem with the Mars Polar Lander
- **Actuality:** attempted to orbit too low and crashed as a result
- **Cause:** metric vs. Imperial conversion in calculations
- **Cost:** US$165m



Other Famous Bugs

- Y2K
- HAL 9000 (2001: A Space Odyssey)
- Estimate that software bugs cost the US economy 0.6% of the GDP
- Over 50% of the development cost of software is on testing and debugging
- No general way of "proving" that a given piece of software implements a given spec

Debugging

- Bugs are inevitable:
  - Fact: even the most carefully-engineered software will include at least 5 errors/1000 lines of code
  - Fact: Windows 10 contained 50–60M lines of code ...
- Bug/error types:
  - **syntax errors** = incompatibility with the syntax of the programming language
  - **run-time errors** = errors at run-time, causing the code to crash
  - **logic errors** = design error, such that the code runs but doesn't do what it is supposed to do
- **Debugging** = the process of systematically finding and fixing bugs

Lecture Summary

- What is PEP8, and what are stylistic conventions to look out for in Python?
- What are best practices for commenting?
- What are bugs and how can we prevent/fix them?

Class Exercise: Spot and Fix the Bugs

Spot and fix the bug(s) in the following code, and classify each as a syntax, run-time or logic error:

```
def substrn(sup, sub)
    '''
    Calculate the number of times `sub`
    occurs in `sup`
    '''
    # pre-calculate length of `sub`
    sub_len = len(Sub)

    # generate all substrings of `sup` of length
    # `sub_len`, and test for identity with `sub`
    for i in range(0, len(sup)-sub_len+1):
        if sup[i:i+sub_len] == sub:
            n += 1

    print("n")
```