Computing in Context: Fall 2024

Lecture 4 | Clean code & debugging

COLUMBIA UNIVERSITY | MAILMAN SCHOOL of PUBLIC HEALTH

Dr Sarah Ashcroft-Jones | Mailman School of Public Health

Clean Code

Aka bad code stinks!

What’s wrong with you?

- You studied the language
- Learnt about all the modules
- Practiced coding for weeks

But your code still feels like a 7 headed apocalyptic monster.

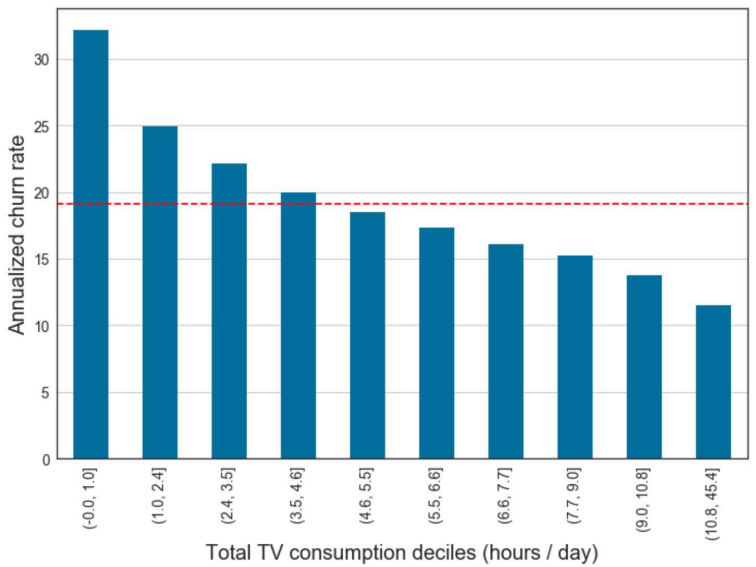
The good news: you can smell it

7.1 Total TV

In [118]:
```
ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

In [119]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f}]'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```



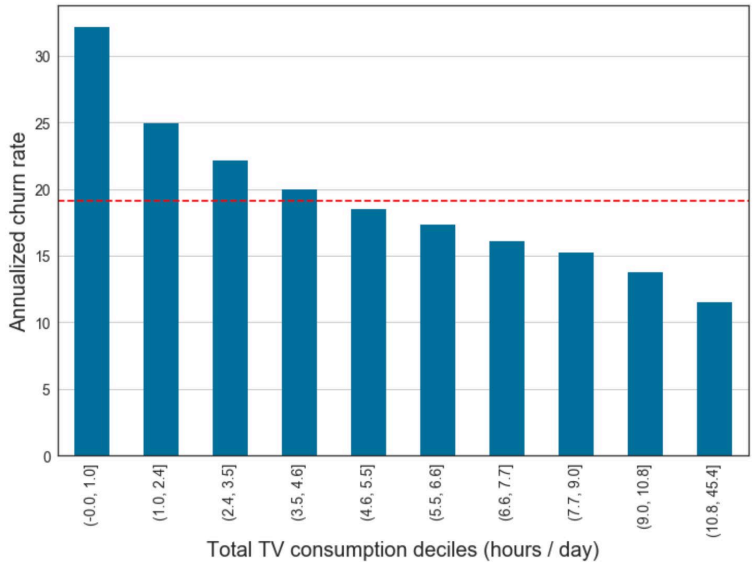
Because ... this stinks!

7.1 Total TV

In [118]:
```
ttv_h_deciles = pd.qcut(ttv_h, 10)
ttv_deciles_churn = tv_merged.groupby(ttv_h_deciles).churned_all.mean() * 12 / 10 * 100
```

In [119]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = ttv_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Total TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f})'.format(x.left, x.right) for x in ttv_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```



7.2 Playback

In [120]:
```
replay_deciles = pd.qcut(tv_merged.PLAYBACK / ttv_sec_to_hpd, 10)
replay_deciles_churn = tv_merged.groupby(replay_deciles).churned_all.mean() * 12 / 10 * 100
```

In [121]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_deciles_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption deciles (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')

    t = ['({:.1f}, {:.1f})'.format(x.left, x.right) for x in replay_deciles_churn.index]
    plt.xticks(range(len(t)), t)
```

In [122]:
```
replay_h = pd.cut(tv_merged.PLAYBACK / ttv_sec_to_hpd, np.arange(0, 8), include_lowest=True)
replay_churn = tv_merged.groupby(replay_h).churned_all.mean() * 12 / 10 * 100
```

In [123]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = replay_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('Replay TV consumption (hours / day)')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

7.3 Trends

In [137]:
```
tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn = tv_merged.groupby(tv_delta).churned_all.mean() * 12 / 10 * 100
```

In [138]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

In [1651]:
```
tv_delta = pd.cut(tv_merged.TTV_201703_delta, np.arange(-5.5, 5.6, 1.0))
tv_delta_churn_some = tv_merged[at_least_some_ttv_mask].groupby(tv_delta[at_least_some_ttv_mas
```

In [1652]:
```
with plt.rc_context(rc=get_style(figsize=(12 ,8))):
    ax = tv_delta_churn_some.plot.bar(color=blue)
    plt.grid(axis='y')
    plt.xlabel('TV consumption trend')
    plt.ylabel('Annualized TV+Vodafone churn rate')
    plt.axhline(annual_tv_churn * 100, c='r', ls='--')
```

What is wrong with smelly code?

It might work right now, but in time will have issues:

- **Hard to read & test**: it is difficult to see an overall structure; understanding the code in one place requires checking the code all over
- **Coupled**: an update in one place requires changes in other places
- **Not flexible**: adding new functionality requires extensive rewrites

Code should be a Lego block structure

Functions (and classes) group together things that go together and form basic blocks.

We can quickly rearrange the blocks to extend a structure or build a new one!

Flexible code is just like a Lego construction:

- is easy to understand in terms of blocks
- tolerates changes
- is reusable

More broadly …

```
>>> 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!
```

https://inventwithpython.com/blog/2018/08/17/the-zen-of-python-explained/

Coding principle 1

DRY (Don't repeat yourself)

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Only rule is that code should not be duplicated.

Instead of duplicating lines, find an algorithm that uses iteration.

Coding principle 2

KISS (Keep it simple, stupid)

Most systems work best if they are kept simple, rather than made complicated.

Simplicity should be a key goal in structuring your code, and unnecessary complexity should be avoided.

Coding principle 3

SoC (Separation of concerns)

> *A design principle for separating a computer program into distinct sections such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.*

For example: create a new module when it makes sense to do so.

But - more modulesmay equal more problems – do you need another?

Clean code in practice

It’s the little things

Naming conventions - variables

Make names easy to understand

```
# This is bad
c = 5
d = 12
```

```
# This is good
city_counter = 5
elapsed_time_in_days = 12
```

Naming conventions - variables

Make names pronounceable

```
from datetime import datetime

# This is bad
genyyyymmddhhmmss = datetime.strptime('04/27/95 07:14:22', '%m/%d/%y %H:%M:%S')
```

```
# This is good
generation_datetime = datetime.strptime('04/27/95 07:14:22', '%m/%d/%y %H:%M:%S')
```

Naming conventions - variables

Avoid weird abbreviations

```
# This is bad
fna = 'Bob'
cre_tmstp = 1621535852
```

```
# This is good
first_name = 'Bob'
creation_timestamp = 1621535852
```

Naming conventions - variables

Avoid “magic” numbers

```python
import random

# This is bad
def roll():
    return random.randint(0, 36)  # what is 36 supposed to represent?
```

```python
# This is good
ROULETTE_POCKET_COUNT = 36

def roll():
    return random.randint(0, ROULETTE_POCKET_COUNT)
```

Commenting conventions

Commenting conventions

1. Don't comment bad code, rewrite it

```
# TODO: RE-WRITE THIS TO BE BETTER
```

Commenting conventions

2. Readable code doesn’t need comments

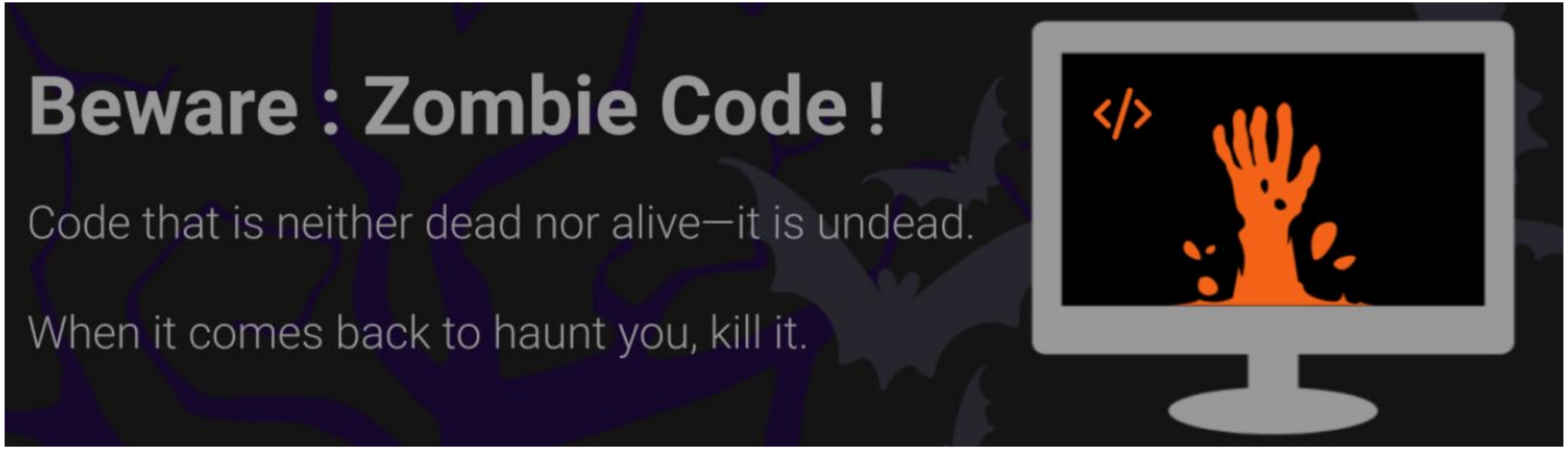
```
# This checks if the user with the given ID doesn't exist.
if not User.objects.filter(id=user_id).exists():
    return Response({
        'detail': 'The user with this ID does not exist.',
    })
```

Commenting conventions

2b. Don't add comments that do not add anything of value to the code.

Commenting conventions

3. Don't leave commented out code



There’s a PEP for that!

There’s a PEP for that!

PEP 8 (Python Enhancement Proposal) a style guide that describes the coding standards for Python.

The most important rules work to guide:
- naming conventions
- line formatting
- whitespace
- comments

There’s a PEP for everything – we could get picky but just be consistent!

Get your PEP on:
https://peps.python.org/pep-0008

1. If you have an acronym in camel case, should you capitalize the first letter only or the whole acronym?
2. How many blank lines go before a function I define at the top of a module? How many after?
3. Which of these is correct: x[3:4] or x[3 : 4]?
4. How many spaces should you use after a period in a comment?
5. Give examples of a long line of code and how you can break it onto multiple lines.

---

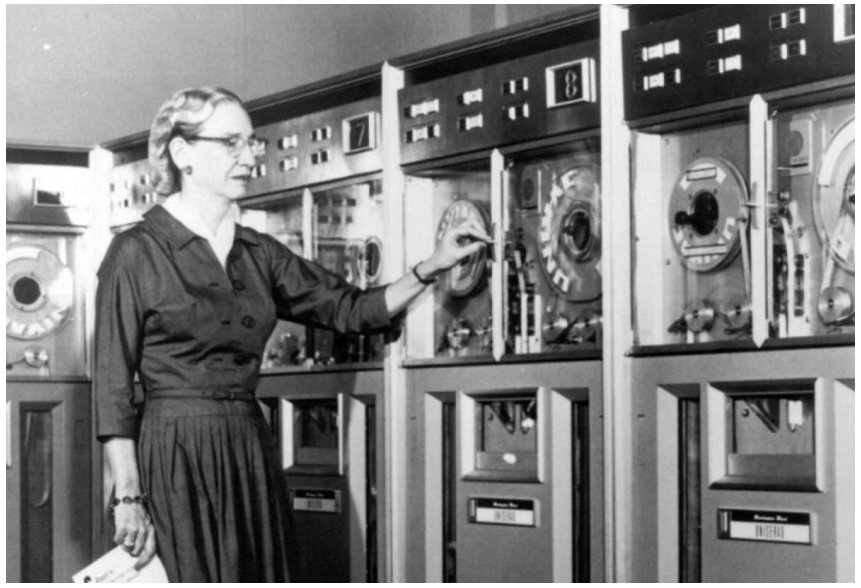
1. What are the capitalization guidelines for classes, functions, and variables? Give examples.
2. Should you use single or double quotes to indicate strings?
3. What is the difference between mixed case, camel case, and snake case?
4. How should you indicate block comments?
5. What are the whitespace rules around operators? Give several examples.

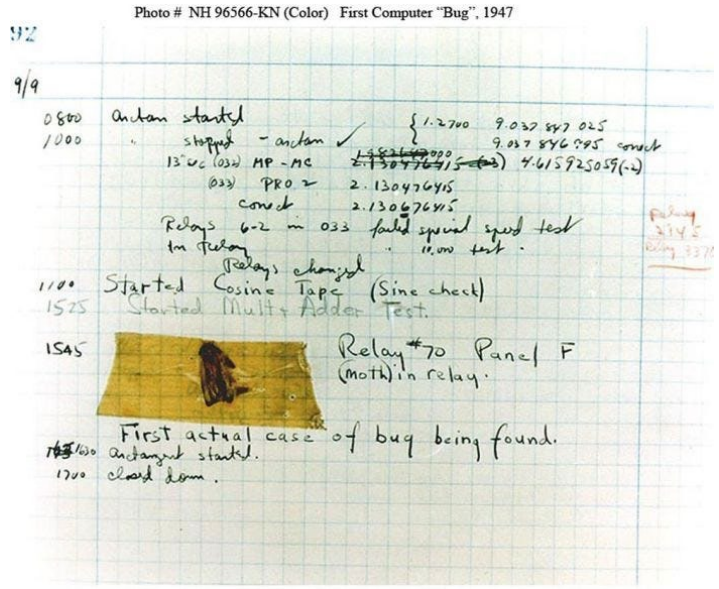
Debugging Code

Your very own Frankenstein’s monster!

Where does “debugging” come from?

Team around Grace Hopper at Harvard found a moth in their computer in 1947 in maybe the first description of a computer bug.





And where do “bugs” come from?

The subtle art of debugging your code

(Sometimes, there is nothing subtle about it – but we can try!)

The subtle art of debugging your code

1. Know when to debug

For some, programming and debugging are the same thing. Programming is the process of gradually debugging a program until it does what you want.

Start with a simple piece of working code and make small incremental modifications, debugging as you go.

If you spend a lot of time debugging, you are writing too much code before you start testing!

The subtle art of debugging your code

2. Know your errors

Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. If there is a syntax error in your program, Python does not run. It displays an error message immediately.

Runtime error: If there are no syntax errors in your program, it can start running. But if something goes wrong, Python errors out (aka an exception) and stops.

Semantic error: Code runs without generating error messages, but it does not do what you intended. Identifying semantic errors can be tricky because it requires you to work backward trying to figure out what it is doing.

The subtle art of debugging your code

3. Know your error messages – what and where?

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

```
  Cell In[49], line 2
    y = 6
    ^
IndentationError: unexpected indent
```

The subtle art of debugging your code

3. Know your error messages – what and where?

```
'126' / 3
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

The subtle art of debugging your code

3. Know your error messages – what and where?

```
1 + 3 / 2
```

```
2.5
```

The subtle art of debugging you code

4. If in doubt – print it out.

Viewing a variable via print()

Checking on a pandas dataframe using .head()

A great way to visually check on things (find those semantic errors!)

The subtle art of debugging you code

5. Build in bug repellant – via testing

```
def uses_any(word, letters):
    """Checks if a word uses any of a list of letters.

    >>> uses_any('banana', 'aeiou')
    True
    >>> uses_any('apple', 'xyz')
    False
    """
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

The subtle art of debugging you code

5. Build in bug repellant – via testing

```
def uses_any(word, letters):
    """Checks if a word uses any of a list of letters.

    >>> uses_any('banana', 'aeiou')
    True
    >>> uses_any('apple', 'xyz')
    False
    """
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

```
run_doctests(uses_any)
```

The subtle art of debugging you code

5. Build in bug repellant – via testing

```
def uses_any_incorrect(word, letters):
    """Checks if a word uses any of a list of letters.

    >>> uses_any_incorrect('banana', 'aeiou') .
    True
    >>> uses_any_incorrect('apple', 'xyz')
    False
    """
    for letter in word.lower():
        if letter in letters.lower():
            return True
        else:
            return False     # INCORRECT!
```

The subtle art of debugging you code

5. Build in bug repellant – via testing

```
run_doctests(uses_any_incorrect)
```

```
**********************************************************************
File "__main__", line 4, in uses_any_incorrect
Failed example:
    uses_any_incorrect('banana', 'aeiou')
Expected:
    True
Got:
    False
```

The subtle art of debugging you code

6. Use an IDE debug mode or explore pdb in the command line…

https://docs.spyder-ide.org/current/panes/debugging.html

The subtle art of debugging you code

7. Keep it small keep it simple (when dealing with big data!)

Scale down the input.

Check summaries and types.

Write self-checks: e.g., if computing an average, check that the result is not greater than the largest element or less than the smallest.

This is a “sanity check” because it detects results that are “insane”.

Sharing Code

Sharing is caring!


GitHub

Questions?
Proposal Memo due Monday at 5pm

COLUMBIA UNIVERSITY | MAILMAN SCHOOL of PUBLIC HEALTH