The how and why of python tools

Josh Borrow
Durham ICC

Durham University

St. Andrews | 15 May 2019

The Situation

- Supervisor has handed you some code that takes in some values and calculates some number
- The code is horrifically slow, and unreadable
- How do we remedy this?

```
#numerical integration

dx = 1.0 / 1000000

output_areas = []

x_cubes = [1, 8, 0.8, 3, 2.0, 77]

for i in range(len(x_cubes)):
    x = 0
    while x < 1.0:
        x = x + dx / 2
        output_areas.append((x_cubes[i] * x**3 +5*x**2 + 2*x + 5) * dx)
        x = x + dx / 2


output = sum(output_areas)
output_areas = []

print x_cubes[i], " gives output"
print output
print
```

What does that do?

```
$ python2 old_python_script.py
1  gives output
7.91667966657

8  gives output
9.66668666645

0.8  gives output
7.86667946657

3  gives output
8.41668166653

2.0  gives output
8.16668066655

77  gives output
26.9167556653
```

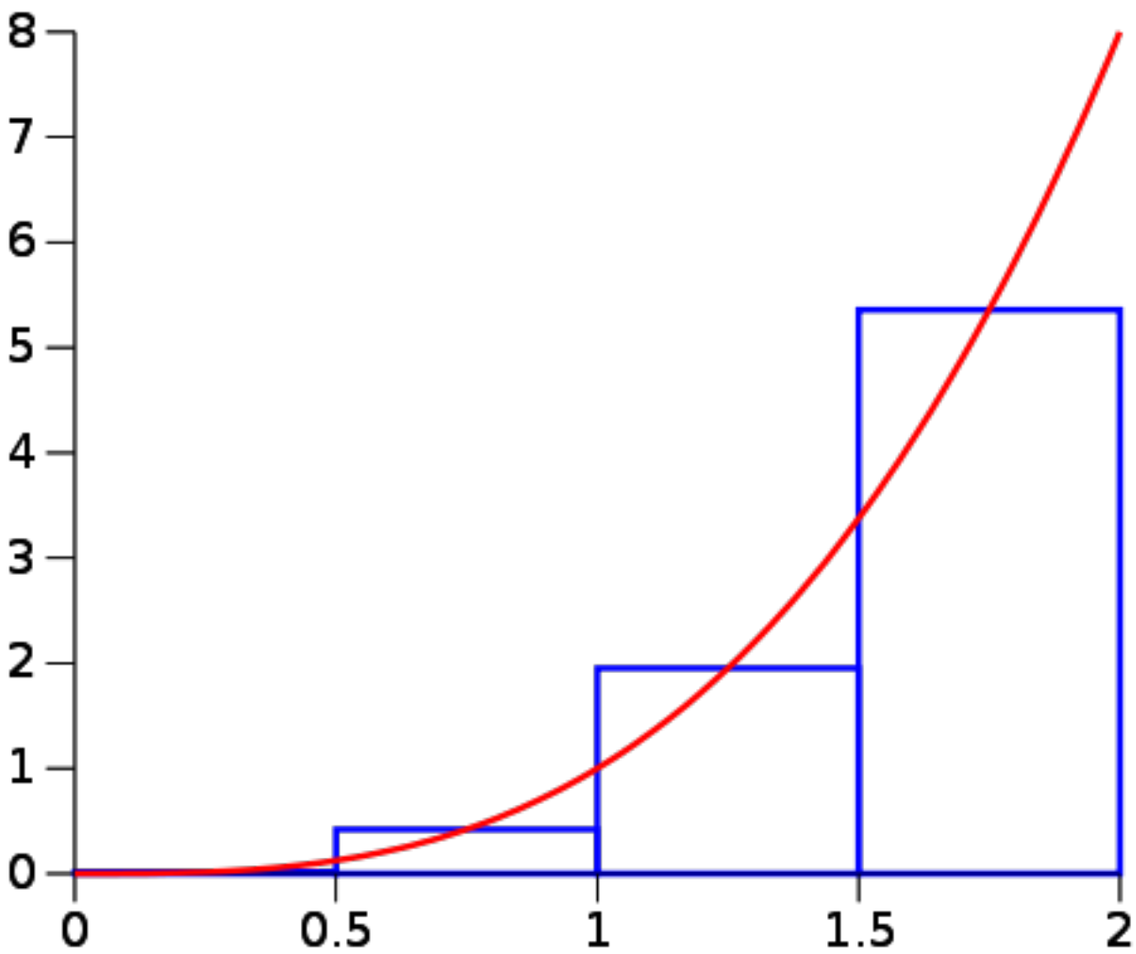
```
#numerical integration

dx = 1.0 / 1000000

output_areas = []

x_cubes = [1, 8, 0.8, 3, 2.0, 77]

for i in range(len(x_cubes)):
    x = 0
    while x < 1.0:
        x = x + dx / 2
        output_areas.append((x_cubes[i] * x**3 +5*x**2 + 2*x + 5) * dx)
        x = x + dx / 2


output = sum(output_areas)
output_areas = []

print x_cubes[i], " gives output"
print output
print
```

These are used as the factor in front of x³

x_cubes $x^3 + 5x^2 + 2x + 5$

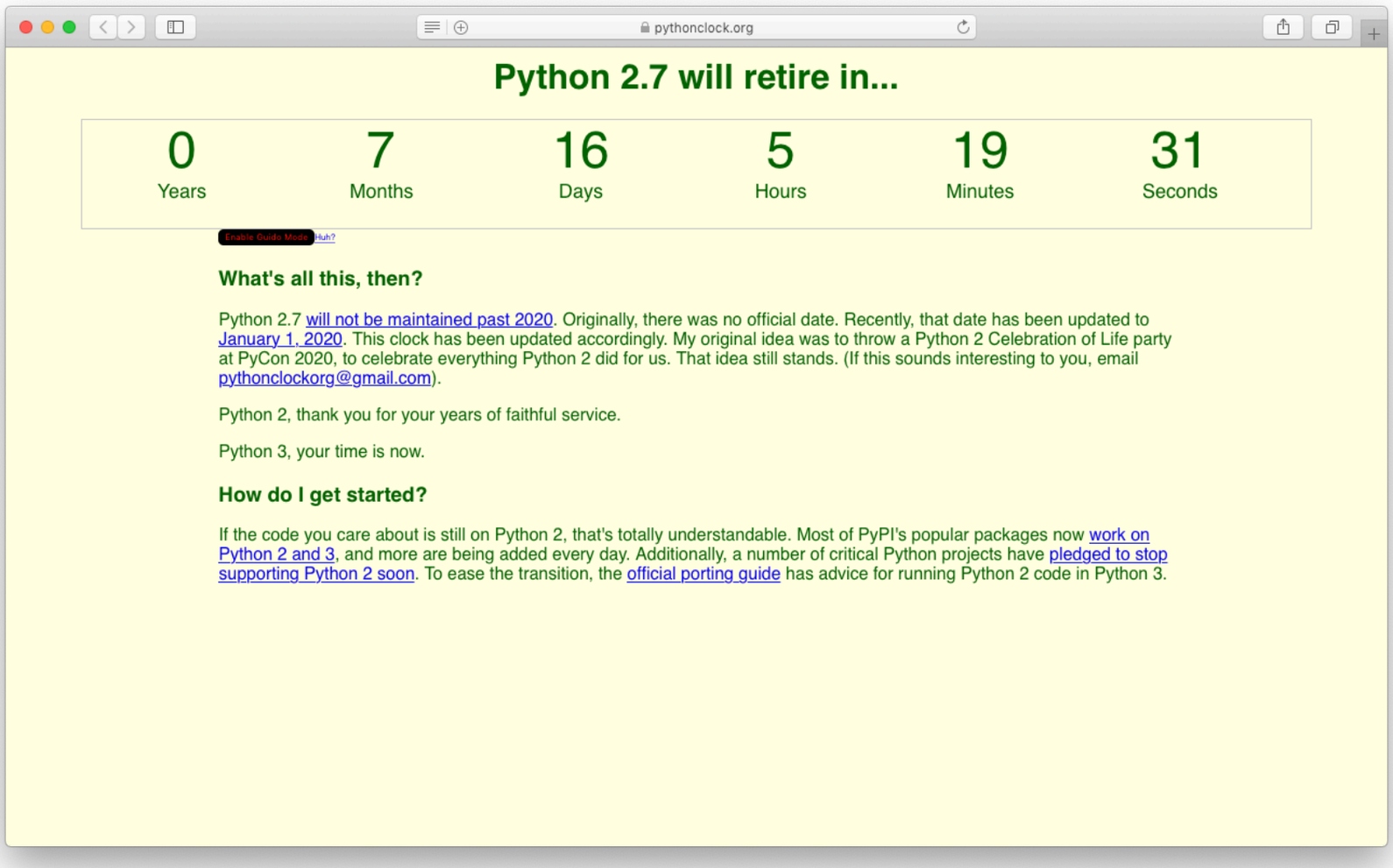
What does that do?

- Looks like the script is numerically integrating a cubic polynomial with a variable factor using the 'rectangular rule'.
- Let's leave aside the fact that this is a completely stupid thing to do and look at how to optimise it.



Conversion to python3

- We should get this script off python2 and move it to python3: https://pythonclock.org



Conversion to python3

- This is super easy! There is a tool called `2to3` for this exact purpose.

```
$ ls
old_python_script.py
$ 2to3 -n -w --add-suffix=3 --no-diffs old_python_script.py
RefactoringTool: Skipping optional fixer: buffer
RefactoringTool: Skipping optional fixer: idioms
RefactoringTool: Skipping optional fixer: set_literal
RefactoringTool: Skipping optional fixer: ws_comma
RefactoringTool: Refactored old_python_script.py
RefactoringTool: Writing converted old_python_script.py to
old_python_script.py3.
RefactoringTool: Files that were modified:
RefactoringTool: old_python_script.py
$ ls
old_python_script.py  old_python_script.py3
```

```python
#numerical integration

dx = 1.0 / 1000000

output_areas = []

x_cubes = [1, 8, 0.8, 3, 2.0, 77]

for i in range(len(x_cubes)):
    x = 0
    while x < 1.0:
        x = x + dx / 2
        output_areas.append((x_cubes[i] * (x**3) +5*x**2 + 2*x + 5) * dx)
    x = x + dx / 2


output = sum(output_areas)
output_areas = []

print(x_cubes[i], " gives output")
print(output)
print()
```

All it changed was the print function

Enter black

- `black` is a python source formatter
- Don't even think about giving it options, just let it do its thing:

```
$ black old_script_py3.py
reformatted old_script_py3.py
All done! ✨ 🍰 ✨
1 file reformatted.
```

- It even comes with free emoji 😍

```
# numerical integration

dx = 1.0 / 1000000

output_areas = []

x_cubes = [1, 8, 0.8, 3, 2.0, 77]

for i in range(len(x_cubes)):
    x = 0
    while x < 1.0:
        x = x + dx / 2
        output_areas.append(
            (x_cubes[i] * (x ** 3) + 5 * x ** 2 + 2 * x + 5) * dx
        )
        x = x + dx / 2

output = sum(output_areas)
output_areas = []

print(x_cubes[i], " gives output")
print(output)
print()
```

Here it just made this easier to read, but it can really help clean up some ugly source files

That can only go so far

- Before we start optimising the code, we need two things:
  - A known good state to compare against, ideally automatically - a unit test - this requires refactoring
  - Some idea of how it performs now to see if we have actually improved it - a profile.

Refactoring

- Changing how the code is written and behaves, without changing the result.

- Just in the same way as you would re-write sections of a paper if it was unclear, why not do the same for code?

- Refactoring can both make code longer or shorter, depending on the situation. Code does not need to be short, it needs to be understandable and fast.

```python
# numerical integration

from typing import List, Tuple


def solve_integrals(
    cube_prefactors: List[float],
    n_steps: int,
    integration_range: Tuple[float],
) -> List[float]:
    """
    Solves the integrals:

    [cube_prefactor] x^3 + 5x^2 + 2x + 5

    using the rectangle rule between the integration bounds given
    with n_steps rectangles. Returns a list of results, one for each
    of the cube_prefactors.
    """

    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    for i in range(len(cube_prefactors)):
        output_areas = []
        x = integration_range[0]
        while x < integration_range[1]:
            x = x + dx / 2
            output_areas.append(
                (cube_prefactors[i] * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )
            x = x + dx / 2

        output = sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
    n_steps = 1_000_000
    integration_range = (0.0, 1.0)
    x_cubes = [1, 8, 0.8, 3, 2.0, 77]

    integrals = solve_integrals(x_cubes, n_steps, integration_range)

    for integral, cube in zip(integrals, x_cubes):
        print(f"{cube} gives output\n{integral}\n")
```

Type hints!

Added a little documentation

Moved code to be in a function, this will allow us to unit test it

Moved constants (and printing) to only be performed in script mode

Used f-strings for clearer printing

```python
# numerical integration

from typing import List, Tuple


def solve_integrals(
    cube_prefactors: List[float],
    n_steps: int,
    integration_range: Tuple[float],
) -> List[float]:
    """
    Solves the integrals:

    [cube_prefactor] x^3 + 5x^2 + 2x + 5

    using the rectangle rule between the integratio
    with n_steps rectangles. Returns a list of resu
    of the cube_prefactors.
```

Type hints!

```python
from typing import List, Tuple


def solve_integrals(
    cube_prefactors: List[float],
    n_steps: int,
    integration_range: Tuple[float],
) -> List[float]:
    """
    Solves the integrals:

    [cube_prefactor] x^3 + 5x^2 + 2x + 5

    using the rectangle rule between the integration bounds given
    with n_steps rectangles. Returns a list of results, one for each
    of the cube_prefactors.
    """

    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    for i in range(len(cube_prefactors)):
        output_areas = []
```

Added a little documentation

```
    with n_steps rectangles. Returns a list of results, one for each
    of the cube_prefactors.
    """

    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    for i in range(len(cube_prefactors)):
        output_areas = []
        x = integration_range[0]
        while x < integration_range[1]:
            x = x + dx / 2
            output_areas.append(
                (cube_prefactors[i] * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )
            x = x + dx / 2

        output = sum(output_areas)
        integrals.append(output)

    return integrals
```

Moved code to be in a function, this will allow us to unit test it

```
            output_areas.append(
                (cube_prefactors[i] * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )
            x = x + dx / 2

        output = sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
    n_steps = 1_000_000
    integration_range = (0.0, 1.0)
    x_cubes = [1, 8, 0.8, 3, 2.0, 77]

    integrals = solve_integrals(x_cubes, n_steps, integration_range)

    for integral, cube in zip(integrals, x_cubes):
        print(f"{cube} gives output\n{integral}\n")
```

Moved constants (and printing) to only be performed in script mode

Used f-strings for clearer printing

```python
# numerical integration

from typing import List, Tuple


def solve_integrals(
    cube_prefactors: List[float],
    n_steps: int,
    integration_range: Tuple[float],
) -> List[float]:
    """
    Solves the integrals:

    [cube_prefactor] x^3 + 5x^2 + 2x + 5

    using the rectangle rule between the integration bounds given
    with n_steps rectangles. Returns a list of results, one for each
    of the cube_prefactors.
    """

    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    for i in range(len(cube_prefactors)):
        output_areas = []
        x = 0
        while x < 1.0:
            x = x + dx / 2
            output_areas.append(
                (x_cubes[i] * x ** 3 + 5 * x ** 2 + 2 * x + 5) * dx
            )
            x = x + dx / 2

        output = sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
    n_steps = 1_000_000
    integration_range = (0.0, 1.0)
    x_cubes = [1, 8, 0.8, 3, 2.0, 77]

    integrals = solve_integrals(x_cubes, n_steps, integration_range)

    for integral, cube in zip(integrals, x_cubes):
        print(f"{cube} gives output\n{integral}\n")
```

Type hints!

Added a little documentation

Moved code to be in a function, this will allow us to unit test it

Moved constants (and printing) to only be performed in script mode

Used f-strings for clearer printing

Check that it works!

```
$ python3 refactored_script.py
1 gives output
7.91667966656792

8 gives output
9.666686666449186

0.8 gives output
7.866679466571464

3 gives output
8.416681666533897

2.0 gives output
8.16668066655114

77 gives output
26.916755665282267
```

```
$ python2 old_python_script.py
1  gives output
7.91667966657

8  gives output
9.66668666645

0.8  gives output
7.86667946657

3  gives output
8.41668166653

2.0  gives output
8.16668066655

77  gives output
26.9167556653
```

Unit tests

- Unit tests automatically test your code for you, using known results.

- Let's create a new file, test_integration.py:

```
from refactored_script import solve_integrals

import numpy as np

def test_integrals():
    n_steps = 1_000_000
    integration_range = (0.0, 1.0)
    x_cubes = [1, 8, 0.8, 3, 2.0, 77]

    integrals = solve_integrals(x_cubes, n_steps, integration_range)
    expected = [7.9166, 9.6666, 7.8667, 8.4166, 8.1667, 26.9167]

    assert np.isclose(integrals, expected, 0.001).all()
```

`pytest`

- `pytest` can be used to run your tests for you, report failures, and drop to the debugger when they go wrong:

```
$ pytest test_integration.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /Users/mphf18/Documents/talks/st_andrews_2019, inifile:
collected 1 item

test_integration.py .                                                    [100%]

========================== 1 passed in 2.99 seconds ===========================
```

Profiling

- Let's see how badly we're doing!

```
$ time python3 refactored_script.py

...

real  0m3.776s
user  0m3.660s
sys   0m0.093s
```

'Obvious' improvements

- Now we have an idea of how quickly the code runs, we can try to improve it.

- Python lists are really slow, right? Let's replace those with numpy arrays.

- We can also calculate some properties ahead of time.

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    for i in range(len(cube_prefactors)):
        output_areas = []
        x = 0
        while x < 1.0:
            output_areas.append(
                (x_cubes[i] * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )
            x = x + dx / 2

        output = sum(output_areas)
        integrals.append(output)

    return integrals
```

Original

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

                                                    Pre-allocate output array
    # These are always the same size
    output_areas = np.empty(n_steps, dtype=float)

    for prefactor in cube_prefactors:                          Change loop
        for step in np.arange(n_steps):                        to use arrays
            x = integration_range[0] + (step + 0.5) * dx
            output_areas[step] = (
                (prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )

        output = np.sum(output_areas)
        integrals.append(output)

    return integrals
```

Use np.sum instead of python sum

Improved

Checks!

```
$ pytest test_integration.py
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /Users/mphf18/Documents/talks/st_andrews_2019, inifile:
collected 1 item

test_integration.py .                                                    [100%]

========================== 1 passed in 19.14 seconds ===========================
```

```
$ time python3 first_improvement.py

...

real 0m22.282s
user 0m24.240s
sys0m4.320s
```

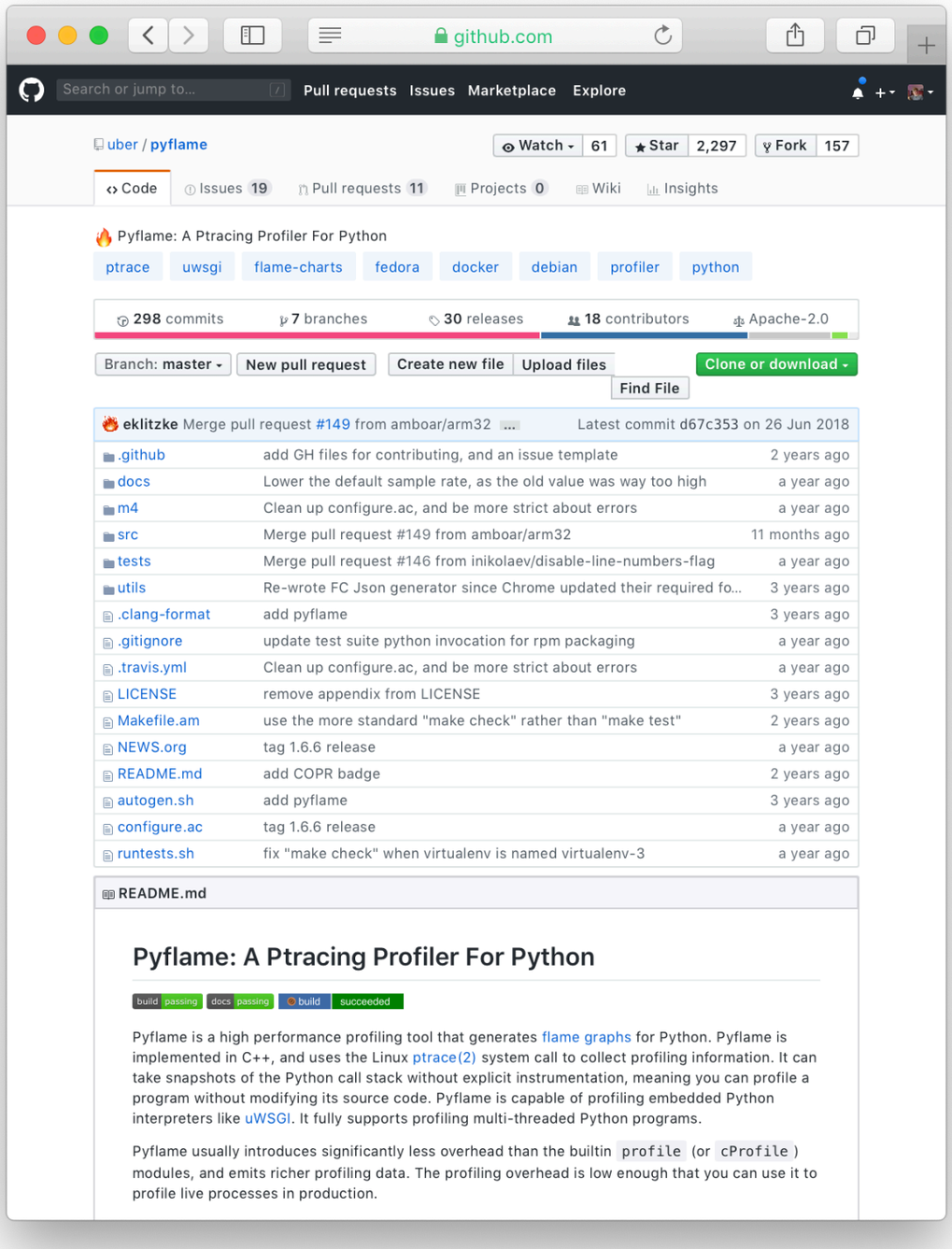
??????????

What went wrong?

- By introducing some 'obvious' optimisations, I have made the code 8x slower.
- How/why did this happen?!??
- We need a more detailed, line-by-line, profile.

Enter Pyflame

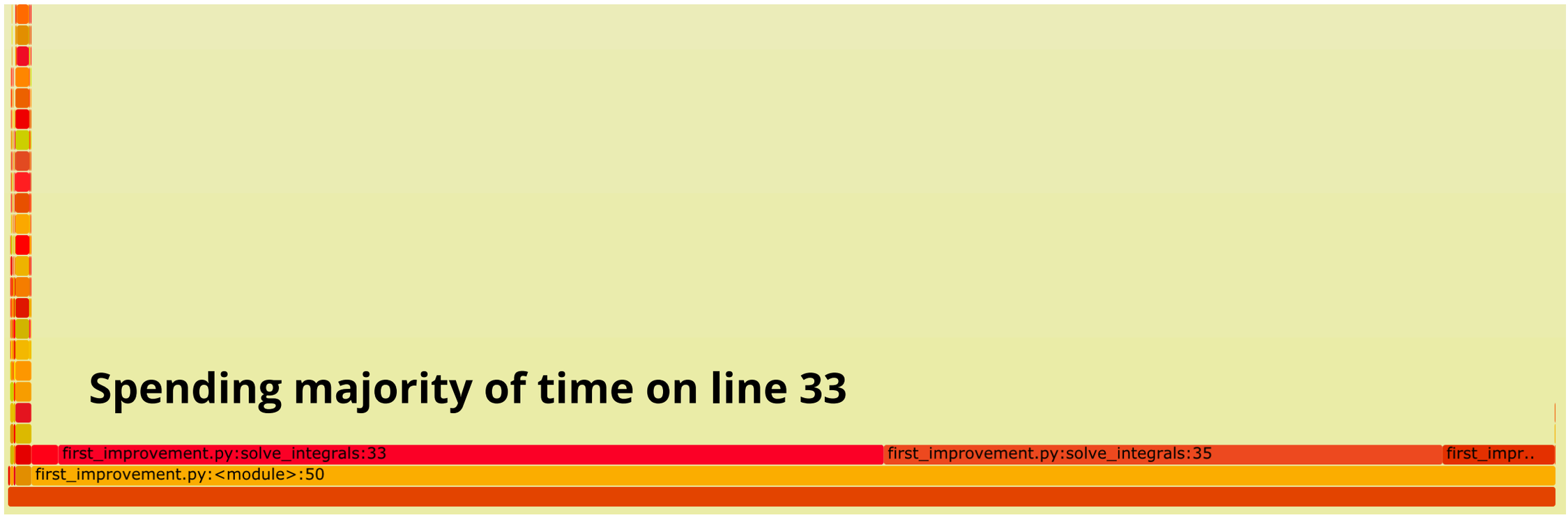
- Written by uber
- A 'sampling' profiler
- Allows you to see line-by-line what's going on in your python program



Let's give it a go!

- Invoke Pyflame with the `--threads` option if you're using numpy as it breaks the 'GIL' at times

```
$ ../pyflame/src/pyflame --threads -o profile_first_improvement.txt -t python3 first_improvement.py
```



```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    output_areas = np.empty(n_steps, dtype=float)

    for prefactor in cube_prefactors:
        for step in np.arange(n_steps):
            x = integration_range[0] + (step + 0.5) * dx
            output_areas[step] = (
                (prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )

        output = np.sum(output_areas)
        integrals.append(output)

    return integrals
```

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    output_areas = np.empty(n_steps, dtype=float)

    for prefactor in cube_prefactors:  np.int64    float
        for step in np.arange(n_steps):   ↓         ↙
            x = integration_range[0] + (step + 0.5) * dx
            output_areas[step] = (
                (prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )

        output = np.sum(output_areas)
        integrals.append(output)

    return integrals
```

Mixed types are bad!

- Mixed types, in any programming language, but especially python, are extremely dangerous.

- They require the creation of a new object for every call. Here, we
  - Notice that we want to add a `np.int64` to a python `float`
  - Decide to give the `float` preference
  - Create new object x, a python `float`
  - Do the expensive conversion from an integer to a float

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    output_areas = np.empty(n_steps, dtype=float)

    for prefactor in cube_prefactors:
        for step in np.arange(n_steps):
            x = integration_range[0] + (step + 0.5) * dx
            output_areas[step] = (
                (prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5)
                * dx
            )

        output = np.sum(output_areas)
        integrals.append(output)

    return integrals
```

Original

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    x = (
        np.arange(n_steps, dtype=np.float32) + 0.5
    ) * dx + integration_range[0]

    for prefactor in cube_prefactors:
        output_areas = (
            prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5
        ) * dx
        output = np.sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
```

Improved

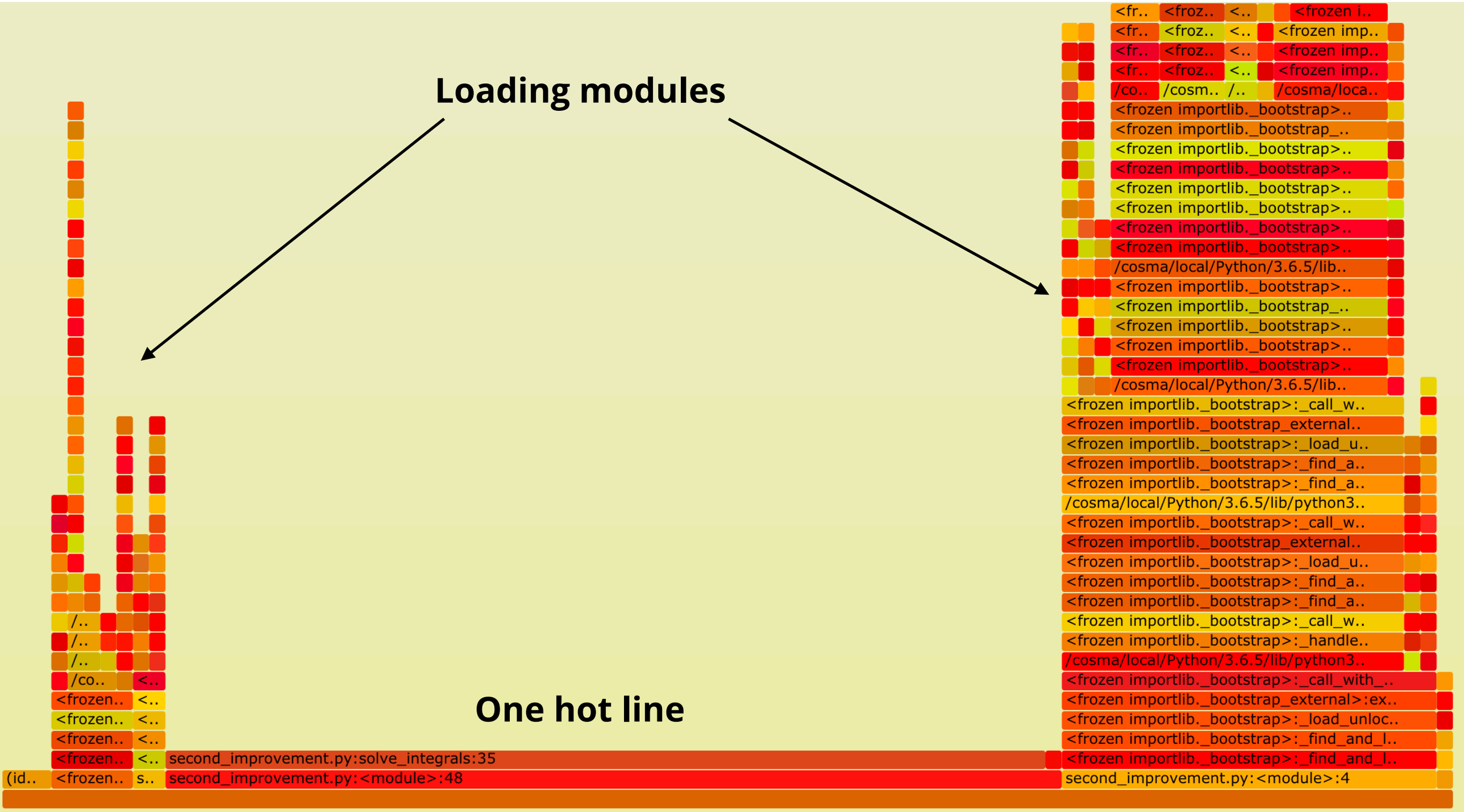
What does the data say?

- We have now vectorised the operation with numpy
- But does this actually make things run faster? Yes!

```
$ time python3 second_improvement.py

...

real  0m0.908s
user  0m2.738s
sys 0m4.368s
```

Now higher than wall-clock because multi-threaded

Flamegraph


Loading modules
One hot line
<fr..
<froz..
<..
<frozen i..
<frozen imp..
/co..
/cosm..
/..
/cosma/loca..
<frozen importlib._bootstrap>..
<frozen importlib._bootstrap_..
/cosma/local/Python/3.6.5/lib..
<frozen importlib._bootstrap>:_call_w..
<frozen importlib._bootstrap_external..
<frozen importlib._bootstrap>:_load_u..
<frozen importlib._bootstrap>:_find_a..
/cosma/local/Python/3.6.5/lib/python3..
<frozen importlib._bootstrap>:_handle..
<frozen importlib._bootstrap>:_call_with_..
<frozen importlib._bootstrap_external>:ex..
<frozen importlib._bootstrap>:_load_unloc..
<frozen importlib._bootstrap>:_find_and_l..
second_improvement.py:solve_integrals:35
(id..
s..
second_improvement.py:<module>:48
second_improvement.py:<module>:4

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    x = (
        np.arange(n_steps, dtype=np.float32) + 0.5
    ) * dx + integration_range[0]

    for prefactor in cube_prefactors:
        output_areas = (
            prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5
        ) * dx
        output = np.sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
```

Arbitrary exponents

- Computing `x**2` is actually much more expensive than computing `x*x`

- We can again use that result to generate `x**3` cheaply

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    x = (
        np.arange(n_steps, dtype=np.float32) + 0.5
    ) * dx + integration_range[0]

    for prefactor in cube_prefactors:
        output_areas = (
            prefactor * x ** 3 + 5 * x ** 2 + 2 * x + 5
        ) * dx
        output = np.sum(output_areas)
        integrals.append(output)

    return integrals


if __name__ == "__main__":
```

Original

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    x = (
        np.arange(n_steps, dtype=np.float32) + 0.5
    ) * dx + integration_range[0]
```

```
    x_squared = x * x
    x_cubed = x_squared * x
    poly = 5 * x_squared + 2 * x + 5

    for prefactor in cube_prefactors:
        output_areas = prefactor * x_cubed + poly
        output = np.sum(output_areas) * dx
        integrals.append(output)

    return integrals
```

Improved

What does the profile show?

```
real0m0.277s
user0m2.405s
sys 0m4.338s
```

New

15x

```
real  0m3.776s
user  0m3.660s
sys   0m0.093s
```

Original

200x on a larger dataset

```
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-4.3.0, py-1.8.0, pluggy-0.9.0
rootdir: /Users/mphf18/Documents/talks/st_andrews_2019, inifile:
collected 1 item

test_integration.py .                                                    [100%]

=========================== 1 passed in 0.17 seconds ===========================
```

The final result

Import from typing

Import numpy

Summary

- We got some awful, python2 only code
- Converted it to python3 automatically with **`2to3`**
- Formatted it with **`black`** so it was more readable
- Refactored it so we could unit test it with **`pytest`**
- Profiled the code, and with **`time`** and **`pyflame`** we managed to get an (at least) 15x speed-up while knowing it still gave the exact same results!

```
    # Actual step size
    dx = (integration_range[1] - integration_range[0]) / n_steps

    # Returned at the end, same length as cube_prefactors
    integrals = []

    # These are always the same size
    x = (
        arange(n_steps, dtype=float32) + 0.5
    ) * dx + integration_range[0]

    x_squared = x * x
    x_cubed = sum(x_squared * x) * dx
    poly = (5 * sum(x_squared) + 2 * sum(x) + 5) * dx

    for prefactor in cube_prefactors:
        output = prefactor * x_cubed + poly
        integrals.append(output)

    return integrals
```