COMP1730/COMP6730
Programming for Scientists

Code Quality & Debugging

Announcements

* Homework 2 marked in Your lab group this week.
* Homework 3 due this Sunday, 24th March, 11:55pm.
* From now on, homeworks will be marked on functionality and code quality.

Lecture outline

* What is “code quality”?
* Debugging

What is code quality and why should we care?

* Writing code is easy – writing code so that you (and others) can be confident that it is correct is not.
* You will always spend more time finding and fixing the errors that you made (“bugs”) than writing code in the first place.
* Good code is not only correct, but helps people (including yourself) *understand* what it does and why it is correct.

* Example 1: the interpreter doesn't care what you call your variables, as long as they are valid names, but giving them meaningful names makes your code easier to read and understand.
* Example 2: python ignores comments, but they can also make code much easier for a human to understand.
* Example 3: We don't *need* to use functions, but if used appropriately they make our code much easier to understand, write and maintain.

(Extreme) example

* What does this function do? Is it correct?

```
def AbC(ABc):
    ABC = len(ABc)
    ABc = ABc[ABC-1:-ABC-1:-1]
    if ABC == 0:
        return 0
    abC = AbC(ABc[-ABC:ABC-1:])
    if ABc[-ABC] < 0:
        abC += ABc[len(ABc)-ABC]
    return abC
```

(Extreme) example – continued

* What does this function do? Is it correct?

```
def sum_negative(input_list):
    '''Sums up all the negative
    numbers in input_list.'''

    total = 0
    for number in input_list:
        if number < 0:
            total = total + number
    return total
```

Aspects of code quality

* Commenting and documentation (today).
* Variable and function naming (today).
* Code organisation (today and later).
* Code efficiency (today and later).

What makes a good comment?

* Raises the level of abstraction: *what* the code does and *why*, not *how*.
  - Except when "how" is especially complex.
* Describe parameters and assumptions
  – python is not a typed language.
* Up-to-date and in a relevant place.
* Don't use comments to make up for poor quality in other aspects (organisation, naming, etc.).
* Good commenting is more important when learning to program and when working with other people.

What makes a bad comment?

* Stating the obvious.

```
x = 5 # Sets x to 5.
```

* Used instead of good naming.

```
x = 0 # Set the total to 0.
```

* Out-of-date, separate from the code it describes, or flat out wrong.

```
# loop over list to compute sum:
avg = sum(the_list) / len(the_list)
```

* More comments than code is (usually) a sign that your program needs to be reorganised.

Function docstring

* A (triple-quoted) string as the first statement inside a function (module, class) definition.
* State the *purpose* and *limitations* of the function, parameters and return value.

```
def solve(f, y, lower, upper):
    '''Returns x such that f(x) = y.
    Assumes f is monotone and that a solution
    lies in the interval [lower, upper]
    (and may recurse infinitely if not).'''
    ...
```

* Can be read by python's `help` function.

Function Docstrings (continued)

* Can formally specify each parameter and the return value.

```
def solve(f, y, lower, upper):
    '''Finds x such that f(x) = y.
    :param f:  a monotonic function with one
        numeric parameter and return value.
    :param y:  integer or float, the value of
        f to solve for.
    ...
    :return:  float, the value of x such that
        f(x) = y.
```

Rules of thumb for commenting

* Do document functions, classes and modules using proper docstrings.
* Don’t use comments as a substitute for good practice in other areas of code quality (organisation, naming, etc.).
* A good starting point is one comment for three to four lines of code (very rough guide).

Good naming practice

* The name of a function or variable should tell you what it does / is used for.
* Variable names should not *shadow* names of standard types, functions, or significant names in an outer scope.

```
def a_fun_fun(int):
    a_fun_fun = 2 * int
    max = max(a_fun_fun, int)
    return max < int
```

(more about scopes in a coming lecture).

* Names can be long (within reason).
  - A good IDE will autocomplete them for you.
* Short names are not always bad:
  - `i` (`j`, `k`) are often used for loop indices.
  - `x`, `y` and `z` are often used for coordinates.
  - best avoided if the scope is large.
* Don't use names that are confusingly similar in the same context.
  - E.g., `sum_of_negative_numbers` vs. `sum_of_all_negative_numbers` – what's the difference?

Code organisation

* Good code organisation
  - avoids repetition;
  - fights complexity by isolating subproblems and encapsulating their solutions;
  - raises the level of abstraction; and
  - helps you find what you're looking for.
* python constructs that support good code organisation are functions, classes (not covered in this course) and modules (later).

Functions

* Functions promote abstraction, i.e. they separate *what* from *how*.
* A good function (usually) does *one* thing.
* Functions reduce code repetition.
  - Helps isolate errors (bugs).
  - Makes code easier to maintain.
* A function should be as general as it can be without making it more complex.

```
def solve(lower, upper):
    '''Returns x such that
    x ** 2 * pi ~= 1. Assumes ...
```

vs.

```
def solve(f, y, lower, upper):
    '''Returns x such that f(x) ~= y.
    Assumes ...
```

Efficiency

Premature optimisation is the root of all evil in programming.

C.A.R. Hoare

* Modern computers usually have enough power to solve your problem, even if the code is not perfectly efficient.
* Programmer time is far more expensive than computer time.
* Code correctness, readability and clarity is more important than optimisation.

When should you consider efficiency?

* For code that is going to run *very* frequently.
* If your program is too slow to run at all.
  A poor choice of algorithm or data structure may prevent your program from finishing, even on small inputs.
* When the efficient solution is just as simple and readable as the inefficient one.

Debugging

What is a “bug”?

We could, for instance, begin with cleaning up our language by no longer calling a bug a bug but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz. with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer’s own creation.

E. W. Dijkstra, 1988

The debugging process

1. Detection – realising that you have a bug.
2. Isolation – narrowing down where and when it manifests.
3. Comprehension – understanding what you did wrong.
4. Correction; and
5. Prevention – making sure that by correcting the error, you do not introduce another.
6. Go back to step *1*.

Syntax Errors

* It is not valid Python Code
* The interpreter will tell you where they are, but you may need to look at the line above or below for a missing bracket, closing string, etc.
* Most good IDEs will highlight syntax errors before you even run the code.

Runtime Errors

* The code is valid Python code - but it's being used to do something Python doesn't know how to do.
* Causes an exception when run (possibly only under certain conditions).
* Learn to read (and understand) Python's error messages. `ZeroDivisionError` is largely self-explanatory, but understand what causes Python to raise an `AttributeError`.

* Semantic errors (logic errors).
  - The code is syntactically valid and runs without error, but *it does the wrong thing* (perhaps only sometimes).
  - To detect this type of bug, you must have a good understanding of what the code is *supposed* to do. Testing can help with this (more in a later lecture).
  - Logic errors are usually the hardest to detect and to correct, particularly if they only occur under certain conditions.
* python allows you to do many things that you never should.

Isolating and understanding a fault

* Work back from where it is detected (e.g., the line number in an error message).
* Find the simplest input that triggers the error.
* Use print statements (or debugger) to see intermediate values of variables and expressions.
* Test functions used by the failing program separately to rule them out as the source of the error.
  - If the bug only occurs in certain cases, these need to be covered by the test set.

Some common errors

* python is not English.

```
if s == 'y' or 'Y':
    ...

if (s == 'y') or ('Y'):
    ...

if (s == 'y') or True:
    ...

if True:
    ...
```

* Limits of floating point numbers (precision *and* range).

* Loop condition not modified in loop.

```
def solve(f, y, lower, upper):
    mid = (lower + upper) / 2
    while math.fabs(f(mid) - y) > 1e-6:
        if f(mid) < y:
            lower = mid
        else:
            upper = mid
    return mid
```

* Off-by-one.

```
k = 1
while k < n:   # < or <= ?
    k = k * 2
return k       # k or k - 1?
```