Australian National University

COMP1730/COMP6730
Programming for Scientists

Code Quality & Debugging

Lecture outline

* What is “code quality”?
* Debugging

Code quality

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.

(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
    i = 0
    while i < len(input_list):
        if input_list[i] < 0:
            total = total + input_list[i]
        i = i+1
    return total
```

Aspects of code quality

1. Commenting and documentation.
2. Variable and function naming.
3. Code organisation (for large programs).
4. Code efficiency (somewhat).

How to 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.

```
    def sum_negative(input_list):
        """Sums up all the negative numbers in input_list.
        Assuming input_list contains only numbers"""
```

* Don’t use comments to make up for poor quality in other aspects (organisation, naming, etc.).

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

Australian National University

How to comment?

- Up-to-date and in a relevant place.

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

- Another example of bad comment:

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

- More comments than code is (usually) a sign that your program needs to be reorganised.
- Good commenting is more important when learning to program and when working with other people.

Australian National University

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.

Some conventions to 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.
    ...
    :returns: float, the value of x such that
    f(x) = y."""
```

Good naming practice

- The name of a function or variable should tell you what it does / is used for.
- Variable names should not *shadow* a 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.
  - `n` (`m`, `k`) are often used for counts.
  - `x`, `y` and `z` are often used for coordinates.
- 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, e.g., by extensive testing.
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*.

Kinds of errors

- Syntax errors: easy to detect
- Runtime errors: not difficult to detect
- Semantic (logic) errors: difficult to detect

Australian National University

Syntax errors

* IDE/interpreter will tell you where they are.

SyntaxError: invalid syntax

```
if spam = 42:
    print('Hello!')
```

IndentationError: unexpected indent

```
print('Hello!')
   print('Howdy!')
```

Runtime errors

* Code is syntactically valid, but you’re asking the python interpreter to do something impossible.
  - E.g., apply operation to values of wrong type, call a function that is not defined, etc.
  - Causes an *exception*, which interrupts the program and prints an error message.
  - Learn to read (and understand) python’s error messages!

Australian National University

Example runtime errors

TypeError: ’str’ object does not support item assignment

```
spam = ’I have a pet cat.’
spam[13] = ’r’
```

IndexError: list index out of range

```
spam = [’cat’, ’dog’, ’mouse’]
print(spam[6])
```

Sematic 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.
  – 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 n is not int:
    ...
if n is (not int):
    ...
```

* Statement in/not in suite.

```
while i <= n:
    s = s + i**2
    i = i + 1
    return s
```

* Precision *and* range of floating point numbers.

* Loop condition not modified in loop.

```
def sum_to_n(n):
    k = 0
    total = 0
    while k <= n:
        total = total + k
    return total
```

* Off-by-one.

```
def smallest_power2(n):
"""Return the smallest power of 2 that is >= n"""
    k = 1
    while k < n:
        k = k * 2
    return k
```

Take home messages

* It’s important to comment, organise your code with good naming, so that others can understand it. Efficiency is only of secondary importance.
* Syntax, runtime and logic errors are 3 kinds of bugs, where logic ones are most difficult to find. Try to design good test cases to debug (more later).