COMP1730/COMP6730
Programming for Scientists

Code Quality & Debugging

Lecture outline

* What is “code quality”?
* Testing & debugging
* Defensive programming

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 often 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):
    """Return sum of all negative numbers in input_list.
    Assumes: list of numerical values. (precondition) """

    total = 0        # cumulative sum
    i = 0            # current list index
    while i < len(input_list):
        if input_list[i] < 0:
            total = total + input_list[i]
        # total now has cumulative sum of negative values
        # for prefix input_list[0,i+1]
        # (loop invariant)
        i = i+1
    return total     # total has cumulative sum
                     # of negatives for input_list
                     # (post-condition)
```

Aspects of code quality

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

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 a dynamically typed language. – (types are not statically specified explicitly)

```
def sum_negative(input_list):
    """Return sum of negative numbers in input_list.
    Assumes input_list contains only numbers."""
```

* Up-to-date and in a relevant place.
* Good commenting is most important when learning to program and when working with other people

How not to comment

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

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

* Wrong, or in the wrong place.

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

* Stating the obvious.

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

* Assume the reader knows python.

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, and side-effects and assumptions (preconditions) if relevant.

```
def solve(f, y, lower, upper):
    """Returns x such that f(x) = y (approximately).
    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 format conventions for parameters and return value:

```
def solve(f, y, lower, upper):
    """Find x such that f(x) = y (approximately).

    :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, value of x such that f(x) within +/- 1e-6 of 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
```

* 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 (covered later in this course) and modules.

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

* This famous quote is often misunderstood.
* You should worry about higher level issues, such as good algorithm design and data structure choice, based on the computational complexity of those algorithms, before complicating your code with micro-optimizations.
* We will discuss computational complexity later in the course.

Efficiency for large scientific and engineering problems

* For scientific and engineering programming, the numpy vector and array library can be an order of magnitude faster than python loops over lists.
* Larger problems may require implementation of core parts in a faster language such as C, or running in parallel on a compute cluster (not covered in this course).
* Profile your code to find time-critical functions before optimising (See python profiler or %timeit in ipython).

Testing & Debugging

Unit testing

* Testing for errors (bugs) in a component of the program – typically a function – is called *unit testing*.
  - Specify the assumptions.
  - Identify test cases (arguments), particularly "edge cases".
  - Verify behaviour or return value in each case.
* The purpose of unit testing is to *detect bugs*.

Good test cases

* Satisfy the assumptions.
* Simple (enough that correctness of the value can be determined “by hand”).
* Cover the space of inputs *and* outputs.
* Cover branches in the code.
* What are edge cases?
  - Integers: 0, 1, -1, 2, 7, 8, 12, 30, ...
  - `float`: very small (1e-308) or big (1e308)
  - Sequences: empty (`''`, `[]`), length one.
  - Any value that requires special treatment in the code.

System testing

* We need to test our scientific and engineering pipelines at multiple levels, including system level.
* System-level tests will typically involve using small real datasets where the true result is known, as well as synthetic test data where the truth is known by construction.
* This stage may also involve measuring performance e.g. for predictive models.

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
  - Easy to detect (when they occur).
  - May be hard to understand (the cause).
* Semantic (logic) errors
  - May be difficult to detect and understand.

Syntax errors

* IDE/interpreter will tell you where they are.

```
  File "test.py", line 2
    if spam = 42:
            ^
SyntaxError: invalid syntax
```

```
if spam == 42:
    print("yes")

  print("spam is:", spam)
```

```
  File "../python/test.py", line 5
    print("spam is:", spam)
                           ^
IndentationError: unindent does not match any outer indentation level
```

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!
  - A problem in dynamically typed languages like python.

```
>>> pets = ['cat', 'dog', 'mouse']
>>> 'I have ' + len(pets) + ' pets'

TypeError: can only concatenate str (not "int") to str

>>> print(pets[3])

IndexError: list index out of range

>> print(pests[0])

NameError: name 'pests' is not defined

>>> print(pets(0))

TypeError: 'list' object is not callable
```

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.
  - Logic errors are usually the hardest to detect and to correct, particularly if they only occur under certain conditions.

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` (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

* Statement in/not in block.

```
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_power_of_2(n):
    """Return the smallest power of 2 that is >= n"""
    k = 1            # start at 0 or 1 ?
    p = 2
    while p <= n:    # < or <= ?
        p = p * 2
        k = k + 1
    return k         # k or k - 1 ?
```

Defensive programming

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

* Write code that is easy to read and well documented.
  - If it's hard to understand, it's harder to debug.
* Make your assumptions explicit, and *fail fast* when they are violated.

Assertions

```
assert test_expression
assert test_expression, "error message"
```

* The `assert` statement causes an runtime error if *`test_expression`* evaluates to `False`.
* Violated assumption/restriction results in an immediate error, in the place where it occurred.
* This is essential in scientific programming.
* Don't use assertions for conditions that will result in a runtime error anyway (typically, type errors). (Modern python allows optional typing- not covered in this course).

```python
# Bad practice (delayed error)

def sum_of_squares(n):
    if n < 0:
        return "error: n is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6

m = ...
k = ...
a = sum_of_squares(m)
b = sum_of_squares(m - k)
c = sum_of_squares(k)
if a - b != c:
    print(a, b, c)
```

```
# Good practice (immediate error)

def sum_of_squares(n):
    assert n >= 0, str(n) + " is negative"
    return (n * (n + 1) * (2 * n + 1)) // 6

m = ...
k = ...
a = sum_of_squares(m)
b = sum_of_squares(m - k)
c = sum_of_squares(k)
if a - b != c:
    print(a, b, c)
```

Explicit vs. implicit

```
# Bad practice (implicit behaviour)

def find_box(color):
    pos = 0
    while robot.sense_color():
        if robot.sense_color() == color:
            return pos
        robot.lift_up()
        pos = pos + 1
```

* What is the loop condition?
* What does `find_box` return if no box of that colour is found?

* Write explicit code, even when python implicitly does the same thing.

```
# Good practice (make it explicit)

def find_box(color):
    pos = 0
    while robot.sense_color() != '':
        if robot.sense_color() == color:
            return pos
        robot.lift_up()
        pos = pos + 1
    return None # colour not found
```

* python allows you to do many things that you never should.
* Don't use obscure "language tricks".

Take-away

* Good code organisation and documentation is important:
  - For others to understand your code.
  - For you to understand what you have done wrong.
* Efficiency, generality and compactness are also good qualities of code, but secondary to clarity.
* Always test, and think about good test cases when you code.
* Debugging is a process of understanding, not trial-and-error.