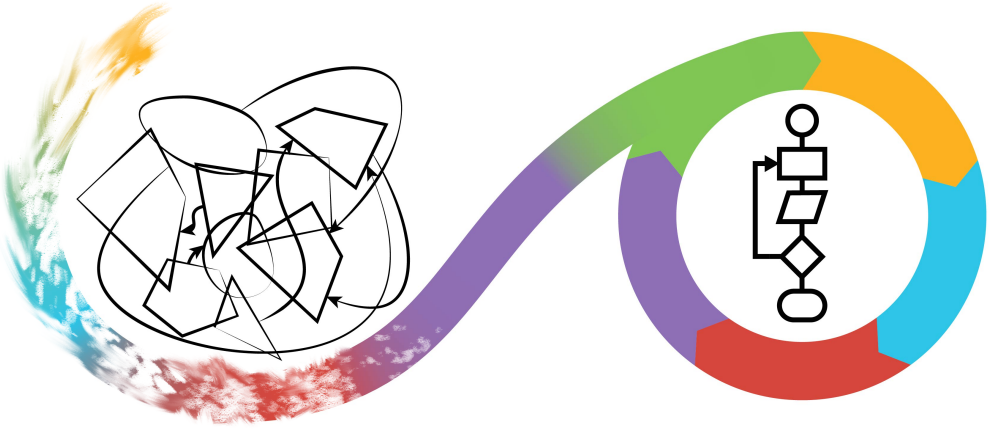
Algorithms

Richèl Bilderbeek

Algorithms 

https://github.com/UPPMAX/programming_formalisms/blob/main/testing/testing_lecture/testing_lecture.qmd



Problem

How do I write functions [1] that are:

- easy to use
- correct

- fast [2]

References;

- [1] For now, we use `algorithm == function`, as the definition of an algorithm is 'a step-by-step procedure for solving a problem or accomplishing some end' [3],
- [2] pick any vague definition
- [3] https://www.merriam-webster.com/dictionary/algorithm

What is a good function?

A good function:

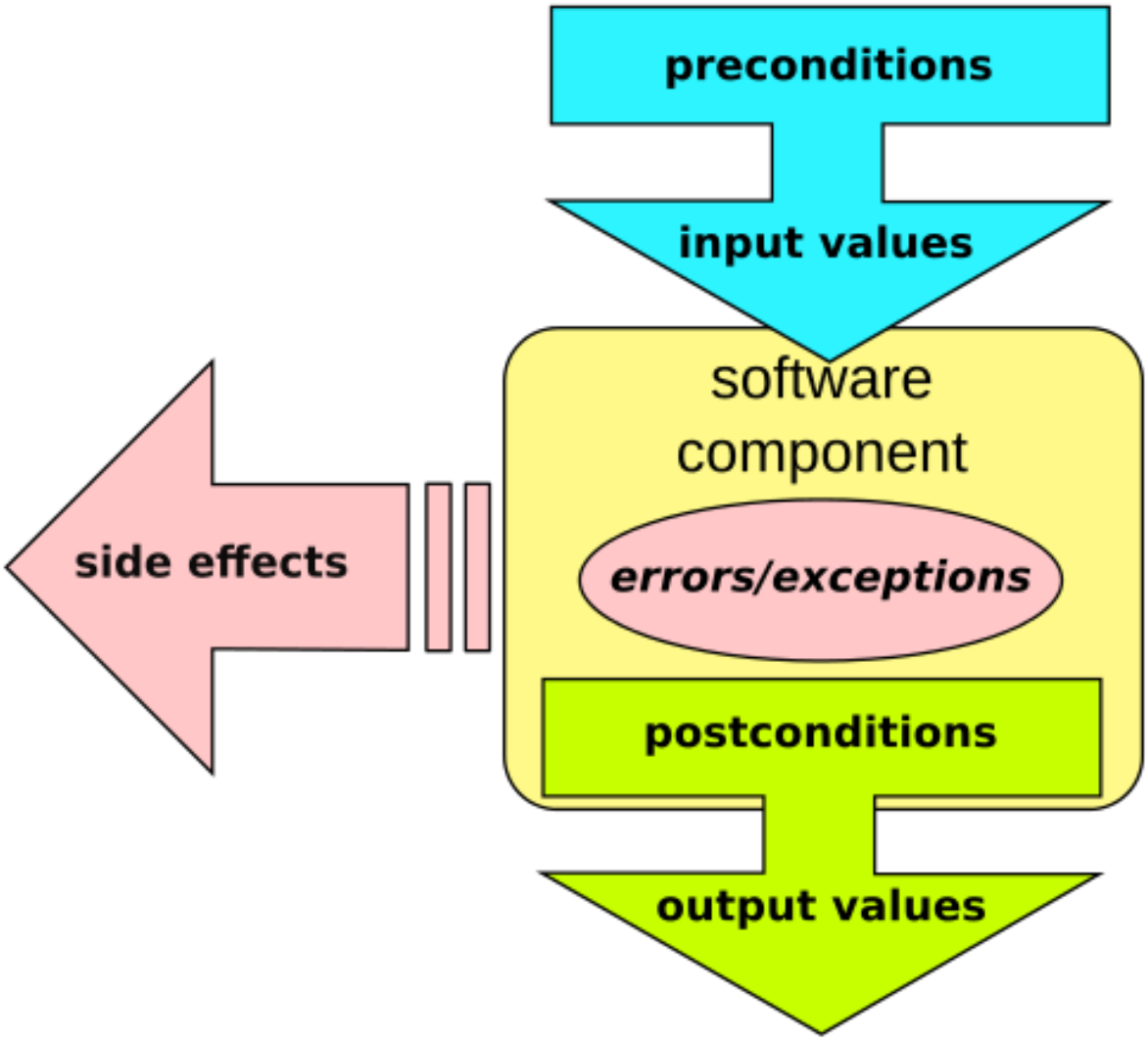
. . .

- Is documented
- Has a clear name
- Has a clear interface
- Does one thing correctly
- Is tested
- Gives clear error messages
- Fast iff needed

References

- C++ Core Guidelines on functions
- R Tidyverse style guidelines on functions
- The Hitchhiker's Guide to Python on general concepts
- (PEP 20: The Zen of Python)

Design by contract



Source: Wikipedia

A good function is documented

```
def sort_1(x):
  """Sort list `x` in-place.

  Returns nothing
  """

def sort_2(x):
  """Sort list `x`.

  Returns the sorted list.
  """

assert sort_1.__doc__ is not None
assert sort_2.__doc__ is not None
```

A good function has a clear name

> There are only two hard things in Computer Science: cache invalidation and naming things
>
> Phil Karlton

Examples of bad function names?

Could you give examples of bad function names?

. . .

- `calculate`: calculates what?

. . .

- `calc_bfgt`: calculates what??

. . .

- `prime`: a prime number is a data type. What does this function do?

. . .

- `needleman_wunch`: this is a technique to get a DNA alignment.

Figure 1: https://www.karlton.org/karlton/images/with-fish.jpg

. . .

Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *        x
```

How would you call the algorithm that detects the location of the `*` (but not of the `x`, as the `*` comes earlier)?

Solutions 1

- `get_first_mismatch_pos`, `get_first_mismatch_locus`, `find_first_mismatch_pos`, `find_first_mismatch_locus`
- you answer that …
    1. starts with a verb
    2. is as English as possible
    3. only uses common abbreviations

Example 2

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *        *
```

How would you call the algorithm that detects all the locations of the `*`s?

Solutions 2

- `find_mismatch_positions`, `find_mismatch_loci`, `get_mismatch_positions`, `get_mismatch_loci`
- you answer that …
    1. starts with a verb
    2. is as English as possible
    3. only uses common abbreviations

Example 3

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

How would you call the algorithm that makes the sequences have as much similarities as possible, by possibly inserting a `-`

```
AAACCCGGGTTT
ATACC-GGGTTT
```

Solutions 3

- `align_seqs`, `align_sequences`, `align_dna_sequences`, `align_dna_seqs`, `calc_aligned_seqs`, `get_aligned_seqs`
- you answer that …
    1. starts with a verb
    2. is as English as possible
    3. only uses common abbreviations

A good function has a clear name

- F.1: "Package" meaningful operations as carefully named functions
- Use verbs, strive for names that are concise and meaningful

A function has a clear interface 1/3

Comment on this function from Pythonpool:

```
i=2

def Prime(no, i):
    if no == i:
        return True
    elif no % i == 0:
        return False
    return Prime(no, i + 1)
```

. . .

Function names start with lowercase character, name does not start with a verb, input is not checked, clumsy interface:

```
assert Prime(2, 2)
assert Prime(3, 2)
assert Prime(3, 3) # Nothing stops me!
assert not Prime(4, 2)
assert Prime(5, 2)
```

The classic on refactoring is (1).

A function has a clear interface 2/3

Comment on this function again:

```
def is_prime(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime(no, i + 1)
```

. . .

- Clumsy interface:

```
assert is_prime(2)
assert is_prime(2, 2) # Nothing stops me!
```

```
assert is_prime(3)
assert not is_prime(4)
assert is_prime(5)
```

A function has a clear interface 3/3

Comment on this function again:

```
def is_prime(no):
    if not isinstance(no, int):
        raise TypeError("'no' must be integer")
    return is_prime_internal(no)

def is_prime_internal(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime_internal(no, i + 1)

assert is_prime(2)
assert is_prime(3)
assert not is_prime(4)
assert is_prime(5)
```

. . .

I think it is OK, please correct me :-)

A function does one thing correctly

F.2: A function should perform a single logical operation, hence don't:

```
def do_x_and_y(): pass

do_x_and_y()
```

Do:

```
def do_x(): pass

def do_y(): pass

do_x()
do_y()
```

You rarely need `and` in a function name. Possible exception: mean and standard deviation

What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- Is tested
- Gives clear error messages
- Is documented
- Fast iff needed

A good function is tested

- F.2: A function should perform a single logical operation: A function that performs a single operation is simpler to understand, test, and reuse.
- Joint Strike Fighter Coding Standards, section 3: Testability: Source code should be written to facilitate testability

Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

The function `align_dna_seqs` aligns two DNA sequences to this:

```
AAACCCGGGTTT
ATACC-GGGTTT
```

Which tests would you write?

Solutions 1

```
assert align_dna_seqs(
  "AAACCCGGGTTT",
  "ATACCCGGGTAT"
  ) == {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
assert align_dna_seqs(
  {
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  }
) ==
  {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
```

```
expect_equal(
  align_dna_seqs(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  ),
  c(
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
expect_equal(
  align_dna_seqs(
    c(
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    )
  ),
  c(
```

```
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
```

What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- **Is tested**
- Gives clear error messages
- Is documented
- Fast iff needed

The role of `assert` within functions

Within functions, `assert` is used for:

- as a stub
- to do things in debug mode only
- to document assumptions a developer makes

`assert` differs between debug and release

```
$ cat assert.py
assert 1 == 2

$ python -O assert.py

$ python assert.py
Traceback (most recent call last):
  File "/home/richel/assert.py", line 1, in <module>
    assert 1 == 2
AssertionError
```

`assert` as a stub

```
def align(dna_sequences):
    """Align the DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

`assert` in debug mode

```
def align_two_dna_sequences(dna_sequences):
    """Internal function to align two DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

Superior to documentation, as it cannot be ignored.

'Assert liberally to document internal assumptions and invariants' (2) chapter 68.

`assert` to document assumptions a developer makes

```
def align_two_dna_sequences(dna_sequences):
    """Align the DNA sequences"""
    # ....
    results = ["AAAA", "AAC-"] # Should be result of calculation
    assert len(results[1]) == len(results[2])
```

Recursive algorithms

- Iterative: use a for-loop
- Recursive: a function that calls itself

n	n!

Example 1: factorial

n	n!
0	1
1	1
2	2 * 1 = 2
3	3 * 2 * 1 = 6
4	4 * 3 * 2 * 1 = 24
5	5 * 4!
n	n * (n-1)!

Exercise 1: factorial

- Develop a function to get the factorial of a number
- If the function used a for-loop, create another function that uses recursion, or the other way around

```
assert calc_factorial_iterative(13) ==
  calc_factorial_recursive(13)
```

```
expect_equal(
  calc_factorial_iterative(13),
  calc_factorial_recursive(13)
)
```

Example 2

Fibonacci sequence:

N	0	1	2	3	4	5	6	7	8	9	10
Fn	0	1	1	2	3	5	8	13	21	34	55

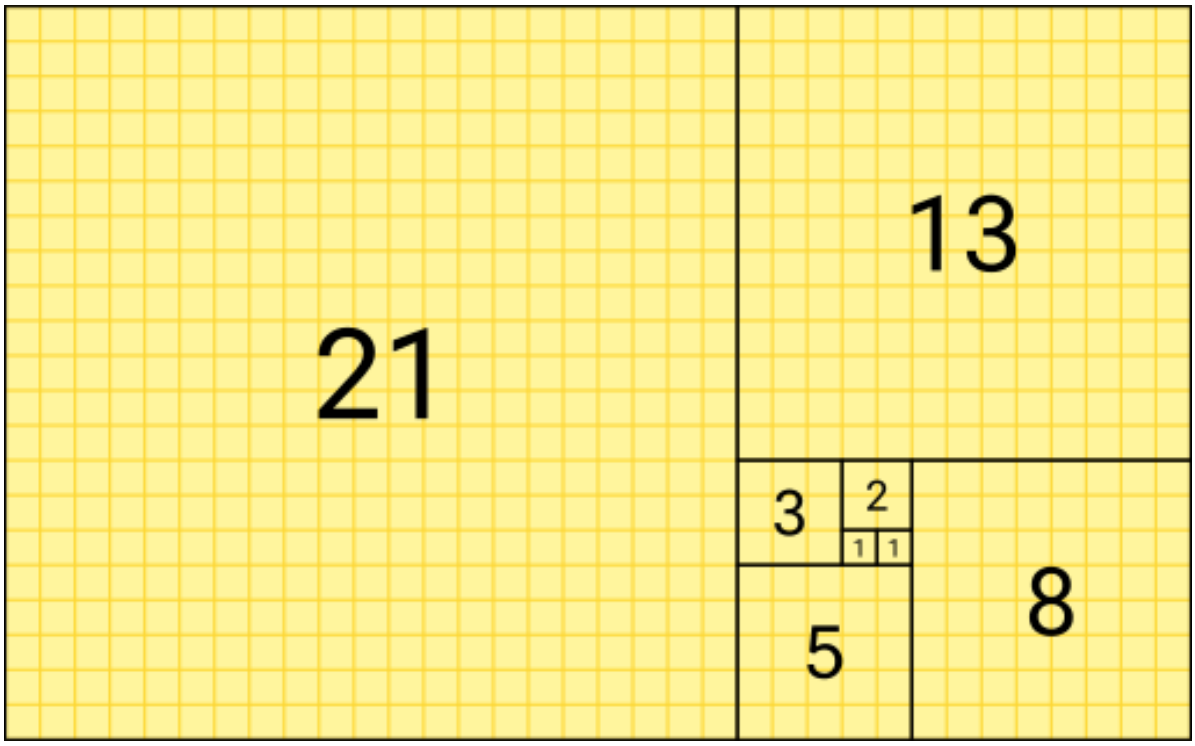


Figure 2: https://en.wikipedia.org/wiki/File:Fibonacci_Squares.svg

Example 2

N	Fn
0	0
1	1
2	1
3	Fn(1) + Fn(2)
n	Fn(n - 1) + Fn(n - 2)

Exercise

- Develop a function to get the nth value in the Fibonacci sequence
- If the function used a for-loop, create another function that uses recursion, or vise versa

```
assert get_nth_fibonacci_iterative(13) ==
  get_nth_fibonacci_recursive(13)
```

```
expect_equal(
  get_nth_fibonacci_iterative(13),
  get_nth_fibonacci_recursive(13)
)
```

Recap

- Function design is hard
- Documentation helps
- TDD helps
- assert helps in debug mode only

Appendix

Links

- Course materials of 2022.

1. Fowler M. Refactoring. Addison-Wesley Professional; 2018.

2. Sutter H, Alexandrescu A. C++ coding standards: 101 rules, guidelines, and best practices. Pearson Education; 2004.