Handling Errors and Debugging

Basic Programming in Python

Sebastian Höffner Aline Vilks

Wed, 3 May 2017

General tips and tricks for surviving this class

- Check your emails[1] as I don't send Stud.IP messages to individuals, only bulk messages. (This is not only useful for this class)
- Try to make your code runnable. It's okay if they have logical errors, but after this lecture (if not already) you are able to read `SyntaxErrors`.
- Sometimes our solutions might give you valuable hints for how problems can be alternatively solved. Or they hint at the next lecture. Or they just give an opportunity to read other people's code.
- Download the slides. They contain some additional note slides which I usually do not present in class. I try to talk about everything but I sometimes forget things, so be sure to check them if you solve the homework.

[1] https://sogo.uos.de/

Homework issues: bitwise operators

What is the difference?

```
a = 6
b = 12
print( a == 6 & b == 12 )
print( a == 6 and b == 12 )
print( (a == 6) & (b == 12) )
```

Output:

```
False
True
True
```

Handling Errors and Debugging

Homework issues: bitwise operators

Homework issues: bitwise operators

What is the difference?

```
a = 6
b = 12
print( a == 6 & b == 12 )
print( a == 6 and b == 12 )
print( (a == 6) & (b == 12) )
```

Output:

```
False
True
True
```

Long story short: use `and` where possible, only use bitwise `&` if you need it.

Bitwise operators have a stronger binding than ==, so `6 & b` is evaluated first. `and` has a weaker binding and is evaluated after ==.

Homework issues: bitwise & (and)

```
a = 0b000110  # 6
b = 0b001100  # 12
print(format(a, '06b'), a)
print(format(b, '06b'), b)
print(format(a & b, '06b'), a & b)
```

Output:

```
000110 6
001100 12
000100 4
```

Homework issues: bitwise | (or)

```
a = 0b000110
b = 0b001100
print(format(a, '06b'), a)
print(format(b, '06b'), b)
print(format(a | b, '06b'), a | b)
```

Output:

```
000110 6
001100 12
001110 14
```

Homework issues: bitwise | (or)

```
a = 0b000110
b = 0b001100
print(format(a, '06b'), a)
print(format(b, '06b'), b)
print(format(a | b, '06b'), a | b)
```

Output:

```
000110 6
001100 12
001110 14
```

For completeness for those really interested:

- `>>` shifts all bits to the right, e.g. `4 >> 1 == 2`
- `<<` shifts to the left: `2 << 1 == 4`
- `^` is the exclusive or (XOR): `6 ^ 12 == 10`
- `~` is the negation, which is a little bit confusing as it starts at the left-most 1 bit.

Homework issues: File I/O, file mode `w`

- Opening a file for reading should be done with mode `r`.
- If you open a file with `w`, it is cleared.
- If you want to avoid clearing but still write, use `a` for appending.
- Or, if you want read and write, use `r+`.

We will usually work with text files, but if we have binary files (e.g. images) we might need `b` as an addition to our mode, e.g. `open(filename, 'rb')`.

Documentation `open()`[2]

[2] https://docs.python.org/3.6/library/functions.html#open

Handling Errors and Debugging

Homework issues: File I/O, file mode `w`

Intermezzo: slides from session 4

Homework issues: File I/O, file mode `w`

- Opening a file for reading should be done with mode `r`.
- If you open a file with `w`, it is cleared.
- If you want to avoid clearing but still write, use `a` for appending.
- Or, if you want read and write, use `r+`.

We will usually work with text files, but if we have binary files (e.g. images) we might need `b` as an addition to our mode, e.g. `open(filename, 'rb')`.

Documentation `open()`[2]

[2] https://docs.python.org/3.6/library/functions.html#open

Homework issues: absolute versus relative paths

Absolute paths specify files from the root directory:

```
/Users/shoeffner/Projects/monty/04_CollectionsFileIO/
        code/hangman_words.txt
C:\Users\shoeffner\Documents\Projects\monty\
        04_CollectionsFileIO\code\hangman_words.txt
```

Relative paths specify files relative to the current working directory:

```
04_CollectionsFileIO/code/hangman_words.txt
hangman_words.txt
```

Homework issues: absolute versus relative paths

Absolute paths specify files from the root directory:

```
/Users/shoeffner/Projects/monty/04_CollectionsFileIO/
        code/hangman_words.txt
C:\Users\shoeffner\Documents\Projects\monty\
        04_CollectionsFileIO\code\hangman_words.txt
```

Relative paths specify files relative to the current working directory:

```
04_CollectionsFileIO/code/hangman_words.txt
hangman_words.txt
```

You can always assume we have the files in the same directory as the scripts (unless otherwise mentioned), so just use their names.

Since my script to generate the slides is not too advanced yet, I have to resort to the slightly longer relative paths as shown on this slide. Sorry for that, but it makes my life much easier at the moment than fiddling around with my automation scripts.

Homework issues: relative paths

For this course, always assume files like the `hangman_words.txt` to be in the same directory as your scripts.

Relative paths:

- Start **not** with a `/` or `\` or `C:\`
- May start with `./` or `../`
- Are mostly system independent (i.e. does not contain `shoeffner` or similar things)
- Are portable

Homework issues: relative paths with \ and /

```
import os
filename = os.path.join('code', 'hangman_words.txt')
print(filename)
```

Output:

```
code/hangman_words.txt
```

Handling Errors and Debugging

Homework issues: relative paths with \ and /

Homework issues: relative paths with \ and /

```
import os
filename = os.path.join('code', 'hangman_words.txt')
print(filename)
```

Output:

```
code/hangman_words.txt
```

For the best portability never use / or \ yourself, but resort to the `os.path` module to `join` paths properly.

However, I use / in the slides for brevity.

Homework issues: variable naming

Do you understand this?

```
def beklemek(ne_kadar=10, nerede='sandalye'):
    numara = 0
    while numara < ne_kadar:
        print(nerede + 'de oturum')
        numara += 1

beklemek(3)
```

Output:

```
sandalyede oturum
sandalyede oturum
sandalyede oturum
```

Handling Errors and Debugging

Homework issues: variable naming

Try to name your variables, write your comments, prints, etc. all in English.

If you want and time allows we can discuss handling different output languages in a future session. But it's not really important for us.

Homework issues: variable naming

Do you understand this?

```
def beklemek(ne_kadar=10, nerede='sandalye'):
    numara = 0
    while numara < ne_kadar:
        print(nerede + 'de oturum')
        numara += 1

beklemek(3)
```

Output:

```
sandalyede oturum
sandalyede oturum
sandalyede oturum
```

Homework issues: variable naming

Can you type this code?

```
ï = 123
ũ = 4125.23
print(ï, ũ)
```

Output:

```
123 4125.23
```

Where on your keyboard are ϕ and π?

Even though sometimes math symbols hold a lot of information, try to use only standard ASCII letters and numbers for your variable names.

Homework issues: variable naming

Can you type this code?

```
ï = 123
ū = 4125.23
print(ï, ū)
```

Output:

```
123 4125.23
```

Where on your keyboard are ϕ and π?

Error messages

```
print 'Hello World!'
```

Output:

```
  File "<string>", line 1
    print 'Hello World!'
                        ^
SyntaxError: Missing parentheses in call to 'print'
```

Reading error messages

```
  File "<stdin>", line 1
    print 'Hello World!'
                        ^
SyntaxError: Missing parentheses in call to 'print'
```

Handling Errors and Debugging

Reading error messages

Reading error messages

```
  File "<stdin>", line 1
    print 'Hello World!'
                       ^
SyntaxError: Missing parentheses in call to 'print'
```

- `File "<stdin>", line 1`: Location in file
- `print 'Hello World!'`: Faulty line
- `^`: Where in the line?
- `SyntaxError`: Error type
- `Missing parentheses in call to 'print'`: Description

Longer error messages

```
def printer():
    print(x)

def caller():
    printer()

caller()
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 7, in <module>
  File "<string>", line 5, in caller
  File "<string>", line 2, in printer
NameError: name 'x' is not defined
```

Handling Errors and Debugging

Longer error messages

- For nested calls, a Traceback is returned
- From top to bottom you can figure out what was called.

Longer error messages

```
def printer():
    print(x)

def caller():
    printer()

caller()
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 7, in <module>
  File "<string>", line 5, in caller
  File "<string>", line 2, in printer
NameError: name 'x' is not defined
```

SyntaxError: Missing parentheses

```
print 'Hello World!'
```

Output:

```
  File "<string>", line 1
    print 'Hello World!'
                        ^
SyntaxError: Missing parentheses in call to 'print'
```

`SyntaxError`: Missing parentheses

```
print('Hello World!')
```

Output:

```
Hello World!
```

SyntaxError: Invalid Syntax

```
print("What is "Python"?")
```

Output:

```
  File "<string>", line 1
    print("What is "Python"?")
                           ^
SyntaxError: invalid syntax
```

SyntaxError: Invalid Syntax

```
print("What is \"Python\"?")
```

Output:

```
What is "Python"?
```

SyntaxError: Unexpected character

```
print("Are you" + \" + "Monty" + \" + "?")
```

Output:

```
  File "<string>", line 1
    print("Are you" + \" + "Monty" + \" + "?")
                                              ^
SyntaxError: unexpected character after line continuation c
```

(Unexpected character after line continuation character)

Handling Errors and Debugging

SyntaxError: Unexpected character

SyntaxError: Unexpected character

```
print("Are you" + \" + "Monty" + \" + "?")
```

Output:

```
  File "<string>", line 1
    print("Are you" + \" + "Monty" + \" + "?")
                                               ^
SyntaxError: unexpected character after line continuation c
```

(Unexpected character after line continuation character)

The line continuation character is \.

`SyntaxError`: Unexpected character

```
print("Are you \"Monty\"?")
```

Output:

```
Are you "Monty"?
```

SyntaxError: EOL[3] while scanning...

```
string = "Hello World!
print(string)
```

Output:

```
  File "<string>", line 1
    string = "Hello World!
                          ^
SyntaxError: EOL while scanning string literal
```

[3] EOL stands for end of line. Also exists for EOF (end of file).

`SyntaxError`: EOL while scanning...

```
string = "Hello World!"
print(string)
```

Output:

```
Hello World!
```

SyntaxError: invalid syntax II

```
import turtle
turtle.shape('turtle'=
turtle.forward(100)
turtle.right(90)
```

Output:

```
  File "<string>", line 4
    turtle.right(90)
          ^
SyntaxError: invalid syntax
```

SyntaxError: invalid syntax II

```
import turtle
turtle.shape('turtle')
turtle.forward(100)
turtle.right(90)
```

Summary `SyntaxError`

`SyntaxErrors` occur whenever you type something Python can't decipher. They are found before the code is actually executed.

Most common causes:

- Missing parentheses
- Missing escape characters or quotes
- Typographical errors

TypeError: object is not callable

```
import random
my_random_number = random()
print(my_random_number())
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: 'module' object is not callable
```

TypeError: object is not callable

```
import random
my_random_number = random.random()
print(my_random_number())
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
TypeError: 'float' object is not callable
```

`TypeError`: object is not callable

```
import random
my_random_number = random.random()
print(my_random_number)
```

Output:

```
0.7770916527570656
```

TypeError: must be X, not Y

```
x = 10
print('I have ' + x + ' bottles')
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: must be str, not int
```

TypeError: must be X, not Y

```
x = 10
print('I have ' + str(x) + ' bottles')
print('I have', x, 'bottles')
```

Output:

```
I have 10 bottles
I have 10 bottles
```

TypeError: X is not iterable

```
numbers = 5
for x in numbers:
    print(x)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: 'int' object is not iterable
```

TypeError: X is not iterable

```
numbers = [5]
for x in numbers:
    print(x)
```

Output:

```
5
```

Summary `TypeError`

`TypeErrors` occur whenever you try something with an object it does not support.

Most common causes:

- Calling a module or variable (i.e. putting parentheses behind it)
- Using a dyadic operator on two different types it does not support
- Using non-iterable types as iterables

Other errors

There is a full list of built-in Python errors[4] in the documentation.

Some important ones you might encounter:

- `IndexError`: You tried to access the wrong elements in a list
- `KeyError`: A dictionary key is not found
- `ZeroDivisionError`: Don't try `1/0`
- `NameError`/`UnboundLocalError`: Something is not yet defined (in the proper scope)

... and many, many more.

[4] https://docs.python.org/3/library/exceptions.html#concrete-exceptions

How to deal with errors?

- Read the error message.
- If you have an idea where it's from, try to fix it.
- Search the web: Search for the exception type, check the documentation, etc.
- If you identified the problem: fix it.
- It happens only in one out of 100 iterations? Great, let's check the debugger!

Handling Errors and Debugging

How to deal with errors?

How to deal with errors?

- Read the error message.
- If you have an idea where it's from, try to fix it.
- Search the web: Search for the exception type, check the documentation, etc.
- If you identified the problem: fix it.
- It happens only in one out of 100 iterations? Great, let's check the debugger!

Despite what everyone tells you: even though there are debuggers (and some of them are great!), most of the time a simple `print` already reveals your problems. Just don't forget to delete it again!

Debugging

- A debugger allows to stop code during its execution
- We can inspect variables after each step!

Interactive Python DeBugger (ipdb)

Figure 1: Spyder debug controls: Run/Pause, execute next line, step in, step out, run to breakpoint, stop

Interactive Python debugger

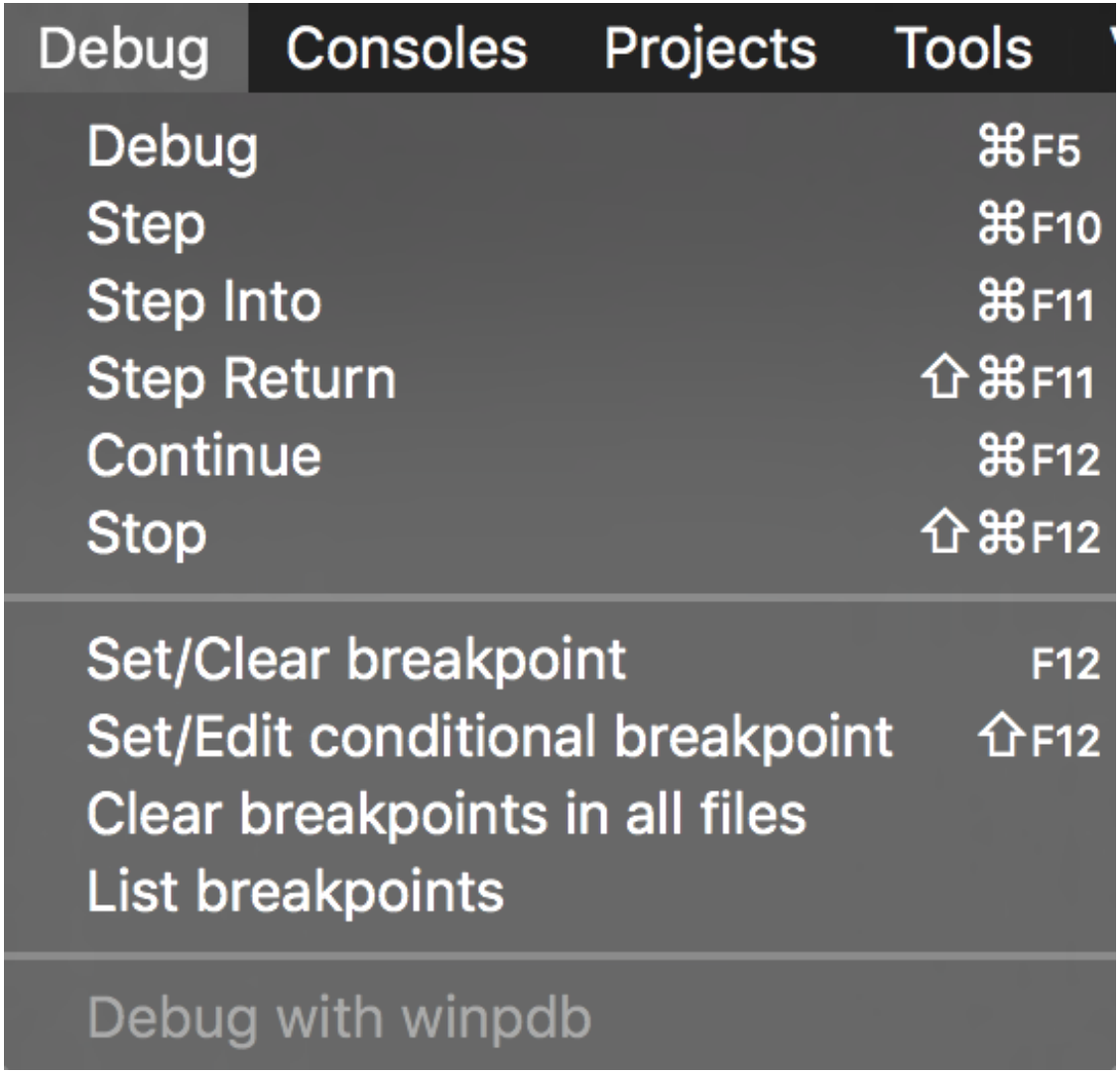


Figure 2: Spyder breakpoint controls

Live demo

File: `code/debug_demo.py`

```
def division(x, y):
    return x / y


result = 0
for i in range(16):
    denominator = i - 10
    result += division(i, denominator)
print(result)
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
  File "<string>", line 2, in division
ZeroDivisionError: division by zero
```

Avoid errors: assertions

Test your code!

```
def add(a, b):
    return a + b

assert add(4, 5) == 9, 'adding 4 and 5 is not 9'
assert add(3, 4) == 7, 'adding 3 and 4 is not 7'
```

Syntax: `assert condition, failmessage`

Handling Errors and Debugging

Avoid errors: assertions

Avoid errors: assertions

Test your code!

```
def add(a, b):
    return a + b

assert add(4, 5) == 9, 'adding 4 and 5 is not 9'
assert add(3, 4) == 7, 'adding 3 and 4 is not 7'
```

Syntax: `assert condition, failmessage`

Use simple examples, complex examples, edge cases... test what you know is correct.

If `condition` is `False`, the test fails and the assertion raises an exception, executing the `failmessage`.

The fail message is optional, but it helps you to figure out, which assertion failed.

Assertions are not always useful: It's not really necessary if you just import a file. But if you do some complex calculations, it is almost always beneficial. Similar to functions, get a feeling when to use them.

Avoid errors: assertions

```
def sub(a, b):
    return a + b

assert sub(5, 4) == 1, '5 - 4 != 1'
assert sub(7, 3) == 4, '7 - 3 != 4'
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
AssertionError: 5 - 4 != 1
```

Avoid errors: assertions

```
def sub(a, b):
    return a - b

assert sub(5, 4) == 1, '5 - 4 != 1'
assert sub(7, 3) == 4, '7 - 3 != 4'
```

Output:

Avoid errors: documentation

4.6. Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

4.6.1. Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations.

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	

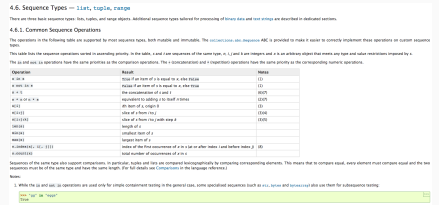
Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see Comparisons in the language reference.)

Notes:

1. While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as `str`, `bytes` and `bytearray`) also use them for subsequence testing:

```
>>> "gg" in "eggs"
True
```

Figure 3: Python 3.6 documentation



4.6. Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

4.6.1. Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The collections.abc.Sequence ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, s and t are sequences of the same type, n, i, j and k are integers and x is an arbitrary object that meets any type and value restrictions imposed by s.

The in and not in operations have the same priorities as the comparison operations. The + (concatenation) and * (repetition) operations have the same priority as the corresponding numeric operations.

Operation	Result	Notes
x in s	True if an item of s is equal to x, else False	(1)
x not in s	False if an item of s is equal to x, else True	(1)
s + t	the concatenation of s and t	(6)(7)
s * n or n * s	equivalent to adding s to itself n times	(2)(7)
s[i]	ith item of s, origin 0	(3)
s[i:j]	slice of s from i to j	(3)(4)
s[i:j:k]	slice of s from i to j with step k	(4)(5)
len(s)	length of s	
min(s)	smallest item of s	
max(s)	largest item of s	
s.index(x[, i[, j]])	index of the first occurrence of x in s (at or after index i and before index j)	(8)
s.count(x)	total number of occurrences of x in s	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see Comparisons in the language reference.)

Notes:

1. While the in and not in operations are used only for simple containment testing in the general case, some specialised sequences (such as str, bytes and bytearray) also use them for subsequence testing:

```
"gg" in "eggs"
True
```

Figure 3: Python 3.6 documentation

Reading documentation will make you a better programmer, as it explains a lot of things.

Imagine you would have to come up with all solutions yourself, or guess what functions do, etc.

Python documentation is usually very elaborate and exhausting, so it's almost always worth to give it a try.

Using documentation

Of course there is a lot of documentation on the web, but take a look at this:

```
def magic():
    """Returns a magic square of size 3x3."""
    return [[2, 7, 6], [9, 5, 1], [4, 3, 8]]

help(magic)
```

Output:

```
Help on function magic in module __main__:

magic()
    Returns a magic square of size 3x3.
```

Using documentation

```
import turtle

help(turtle.up)
```

Output:

```
Help on function up in module turtle:

up()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

Writing documentation

We will roughly follow the Google Python Style Guide[5].

There are others, e.g. Scipy[6] and Python[7] styles, but we use this.

[5] https://google.github.io/styleguide/pyguide.html#Comments

[6] https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt#docstring-standard

[7] https://docs.python.org/devguide/documenting.html

We will roughly follow the Google Python Style Guide[5].

There are others, e.g. Scipy[6] and Python[7] styles, but we use this.

[5] https://google.github.io/styleguide/pyguide.html#Comments
[6] https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt#docstring-standard
[7] https://docs.python.org/devguide/documenting.html

It really does not matter much what you pick, as long as you are consistent throughout a project.

I recommend the Google style because it has the least amount of visual clutter in your code.

I hope to discuss how to build beautiful documentation like the Python docs in a few weeks, the latest when we do the project work.

Writing documentation example

```
def add(left, right):
    """Returns the sum of left and right.

    Args:
        left: The left operand.
        right: The right operand.

    Returns:
        The sum of left and right.
    """
    return left + right
```

Writing documentation explanation

```python
def difficult_function(argument, other_arg=None):
    """Concise description.

    Longer description (if concise is not enough)
    which might need multiple lines.

    Or even some paragraphs.

    Args:
        argument: A description of this argument.
        other_arg: Another description.

    Returns:
        A short summary of what is returned,
        especially its format.

    Raises:
        ValueError: When does this occur?
    """
    pass
```

Writing documentation explanation

```
def difficult_function(argument, other_arg=None):
    """Concise description.

    Longer description (if concise is not enough)
    which might need multiple lines.

    Or even some paragraphs.

    Args:
        argument: A description of this argument.
        other_arg: Another description.

    Returns:
        A short summary of what is returned,
        especially its format.

    Raises:
        ValueError: When does this occur?
    """
    pass
```

You may omit sections (e.g. Args or Returns) if they are irrelevant for you function (not all functions raise nor do all have args).

You can find some documentation in the homework solutions of last week.

More example on how to write it (even for features we have not and will not cover):
http://www.sphinx-doc.org/en/stable/ext/example_google.html

Your fifth homework

- We wrote a little script, but it's horribly broken. Try to fix it and add proper documentation.
- Do some simple (very simple!) data analysis on the famous iris dataset[8].
- From now on: Always document your code!

[8] https://archive.ics.uci.edu/ml/datasets/Iris

The last slide

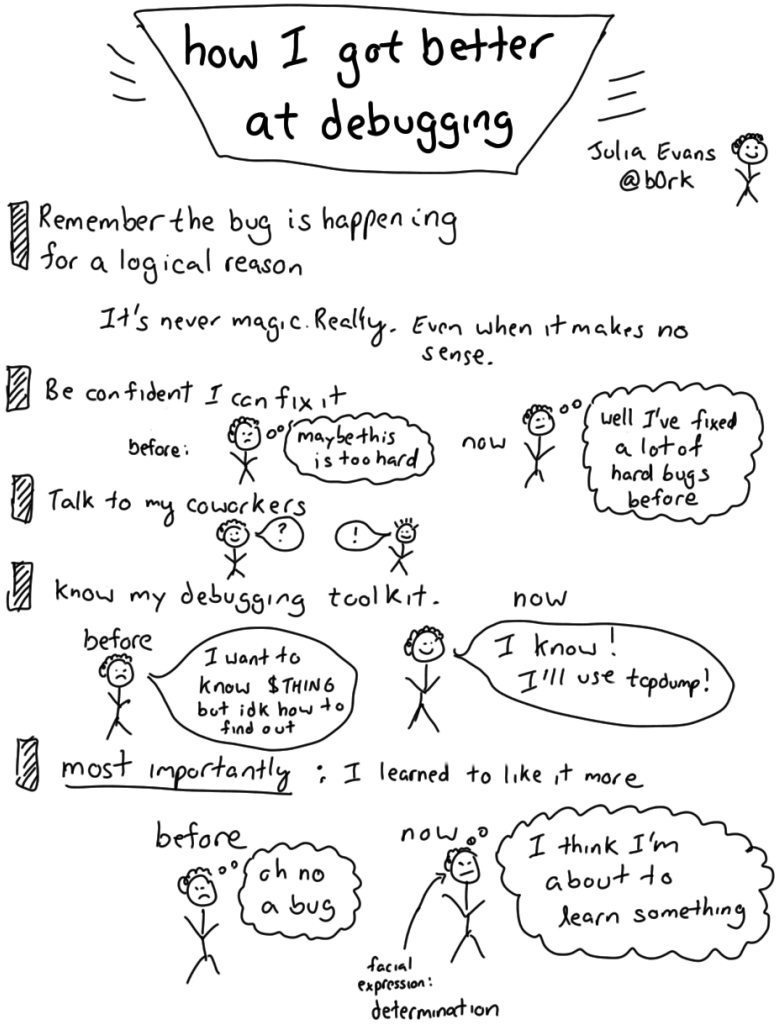


Figure 4: how I got better at debugging (Evans 2016)

References

Evans, Julia. 2016. “How I Got Better at Debugging.” *Julia’s Drawings*, November.