THE UNIVERSITY
of EDINBURGH

Technological Infrastructures for GIS

Organising Python Projects, Testing, Documentation

Magnus Hagdorn

School of GeoSciences
University of Edinburgh

October 7, 2019

THE UNIVERSITY
of EDINBURGH

Introduction

the story so far...

- basic python
- functions
- objects, basic types, container types
- flow control (for, while, if–then–else)
- file I/O
- plotting

Introduction

CC magnus.hagdorn

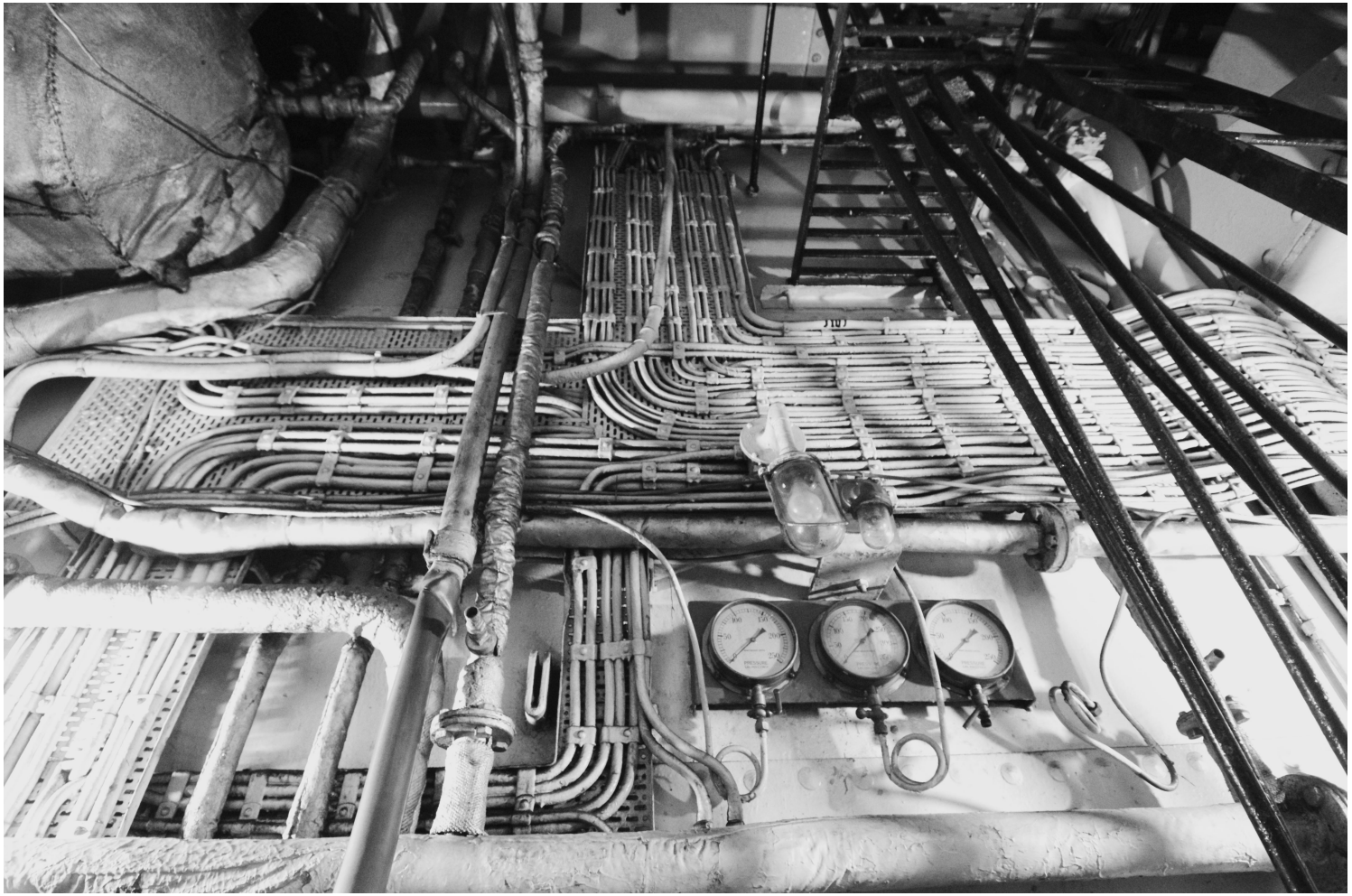
Introduction

Today we will cover

- organising the files of a python project
- exceptions
- debugging
- testing
- documentation

Files, Directories, Modules and Packages

Modules



TIGIS Project
Modules


```
## Simple Scatter Plot
# Load $x,y$ data from a file and plot it.

# setup plotting and load useful modules
import numpy
from matplotlib import pyplot

def loadData2C(fname):
    """load data from a file with two columns"""
    # create empty two lists which will hold the two columns of data
    x = []
    y = []
    # open the file
    inFile = open(fname,'r')
    # loop over each line of the file
    for line in inFile.readlines():
        # split the line and append to the data list
        # we assume that the columns are white-space separated
        # the split method returns a list of strings
        line = line.split()
        x.append(line[0])
        y.append(line[1])
    # return tuple of numpy arrays
    return numpy.array(x,dtype=float),numpy.array(y,dtype=float)

# use the loadData function to read a data file
x,y = loadData2C('some.data')
# plot the data
pyplot.plot(x,y,'o')
# finally show the plot
pyplot.show()
```

Create another file containing

```
#!/usr/bin/env python3

# setup plotting and load useful modules
from matplotlib import pyplot
# also import our own module
import plotData

x,y = plotData.loadData2C("some.data")
# plot the data
pyplot.plot(x,y,'o')
# finally show the plot
pyplot.show()
```

- if you want to, you can make the file executable
  `chmod a+x plot.py`

Problem

All statements in the `plotData` module get executed.

```
if __name__ == '__main__':
    # use the loadData function to read a data file
    x,y = loadData('some.data')
    # plot the data
    pyplot.plot(x,y,'o')
    # finally show the plot
    pyplot.show()
```

Files, Directories, Modules and Packages

Packages

Splitting your project over a number of files has some advantages

- easier to find your way around it
- easier to share with others
- easier to use version control

Version Control

- allows you to track changes of your files
- allows you to collaborate with others
- checkout the University's gitlab service
- or an external service such as github or bitbucket

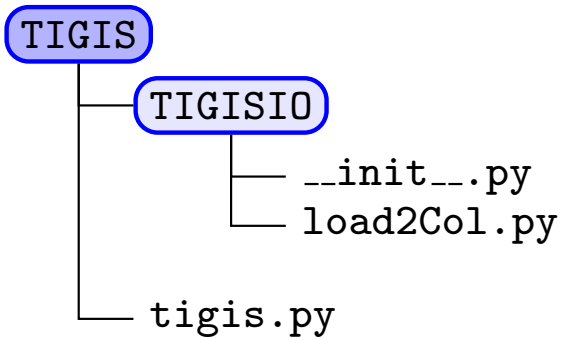
Files, Directories, Modules and Packages

Use Version Control!

[Public domain], via Wikimedia Commons

Files, Directories, Modules and Packages

Packages



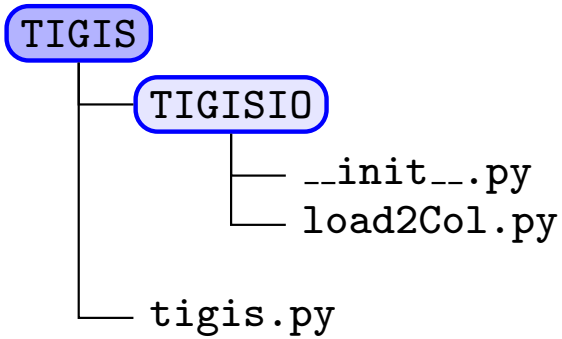
load2Col.py

contains `loadData2C` function:

```
import numpy
__all__ = ['loadData2C']
def loadData2C(fname):
```

Files, Directories, Modules and Packages

Packages



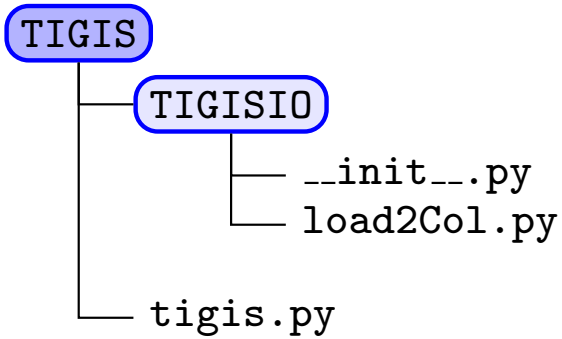
__init__.py

- this is what makes the directory a python package
- the code gets executed when the package is imported

```
from .load2Col import *
```

Files, Directories, Modules and Packages

Packages



tigis.py

```
#!/usr/bin/env python3

# setup plotting and load useful modules
from matplotlib import pyplot
# also import our own module
import TIGISIO
import sys

# get data name from command line arguments
if len(sys.argv) > 1:
    dName = sys.argv[1]
else:
    dName = "../some.data"

x,y = TIGISIO.loadData2C(dName)
# plot the data
pyplot.plot(x,y,'o')
# finally show the plot
pyplot.show()
```

Exercise

- setup directory structure
- create `TIGISIO` package containing
  - `__init__.py`
  - `load2Col.py`
- write a script that uses your `TIGISIO` package
- if you have it already add your I/O function for the programming task

Dealing with Errors and Exceptions

Exceptions

Things will go wrong.
From the Zen of Python we have

- Errors should never pass silently.
- Unless explicitly silenced.

Exceptions

Exceptions

Exceptions are errors detected during the execution of the program

Let's create one, in your ipython console:

```
In [1]: 1/0
Traceback (most recent call last):
<ipython-input-1-05c9758a9c21> in <module>()
----> 1 1/0
ZeroDivisionError: division by zero
```

Exceptions
Handling Exceptions


```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops. That was no valid number. Try again...")
```

- execute the `try` clause
- if no exception occurs the try statement is finished
- if an exception occurs the rest of the try statement is skipped and the except clause is executed
- you can match different exception
- if no exception is matched it is *unhandled* and execution stops

Exceptions
Handling Exceptions

You can access the conent of the exception and do something with it:

```
def div(a,b):
    return a/b

try:
    ratio = div(1,0)
except ZeroDivisionError as detail:
    print ('Handling run-time error: {}'.format(
                                        detail))
else:
    print ('the ratio is: {}'.format(ratio))
```

quite often you would like to tidy up resources *unconditionally*

```
someFile = open('someFile.txt','r')
try:
    someFile.write('hello, world')
except:
    print('that did not work')
finally:
    print('tidying up')
    someFile.close()
print('is file closed: {}'.format(someFile.closed))
```

this is pattern is so common that

```
with open('someFile.txt','r') as someFile:
    someFile.write('hello, world')
print('is file closed: {}'.format(someFile.closed))
```

- other resources such as locks also support context managers

Exceptions
Raising Exceptions

When you know that something went wrong you can indicate this by *raising an exception:*

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError:  HiThere
```

have a look at the list of built-in exceptions.

Exceptions

Task

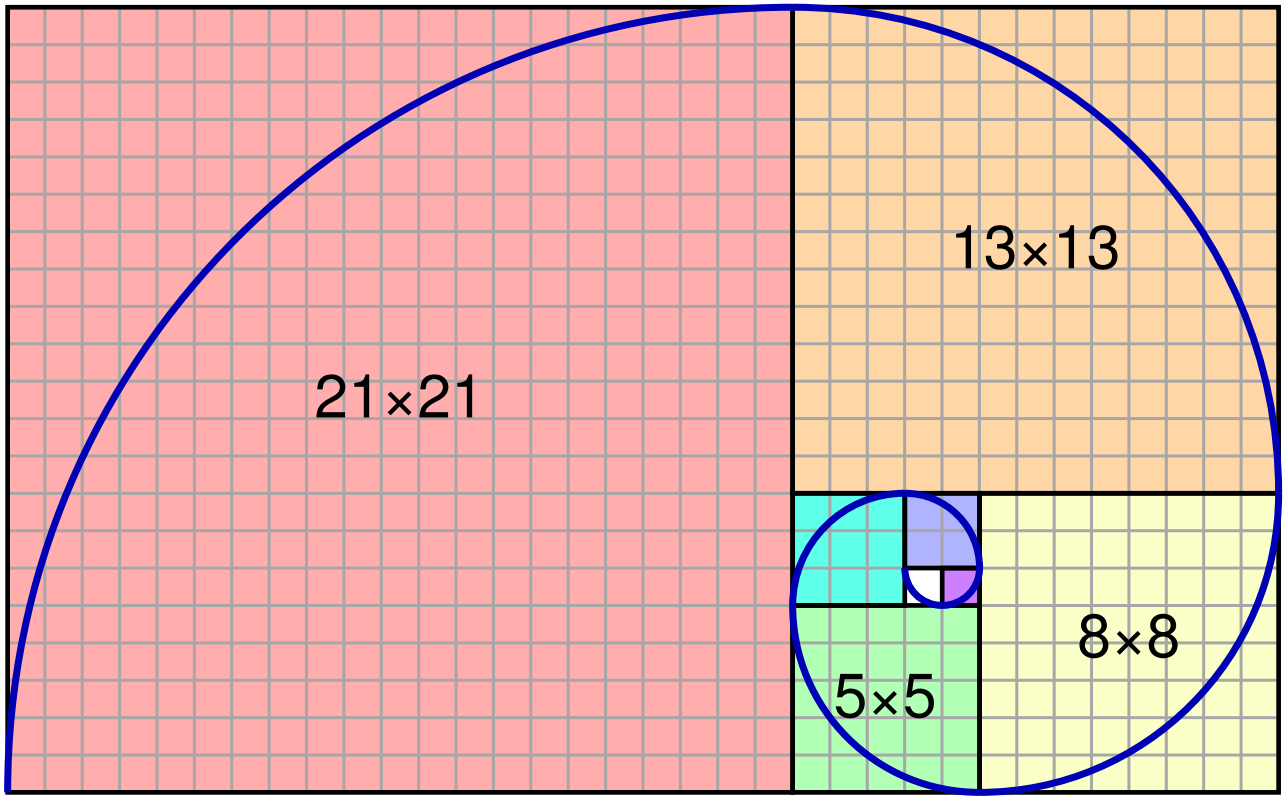
Exercise

- play with exceptions and raising them
- use your data reader to read a broken data file (you can break one of the scatter file by adding some lines that are not numbers)
- use a try/except block to print out a warning when you cannot read a line rather than fail
- use a context manager for your data reader

Debugging

Finding errors

Debugging
Fibonacci Numbers



CCO Jahobr

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

with $F_1 = F_2 = 1$
Implement using recursion:

```
def fibr(n):
    if n<1:
        raise RuntimeError
    if n<3:
        return 1
    else:
        return fibr(n-1) + fibr(n-2)

if __name__ == '__main__':
    print(fibr(10))
```

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

with $F_1 = F_2 = 1$
Implement using a generator:

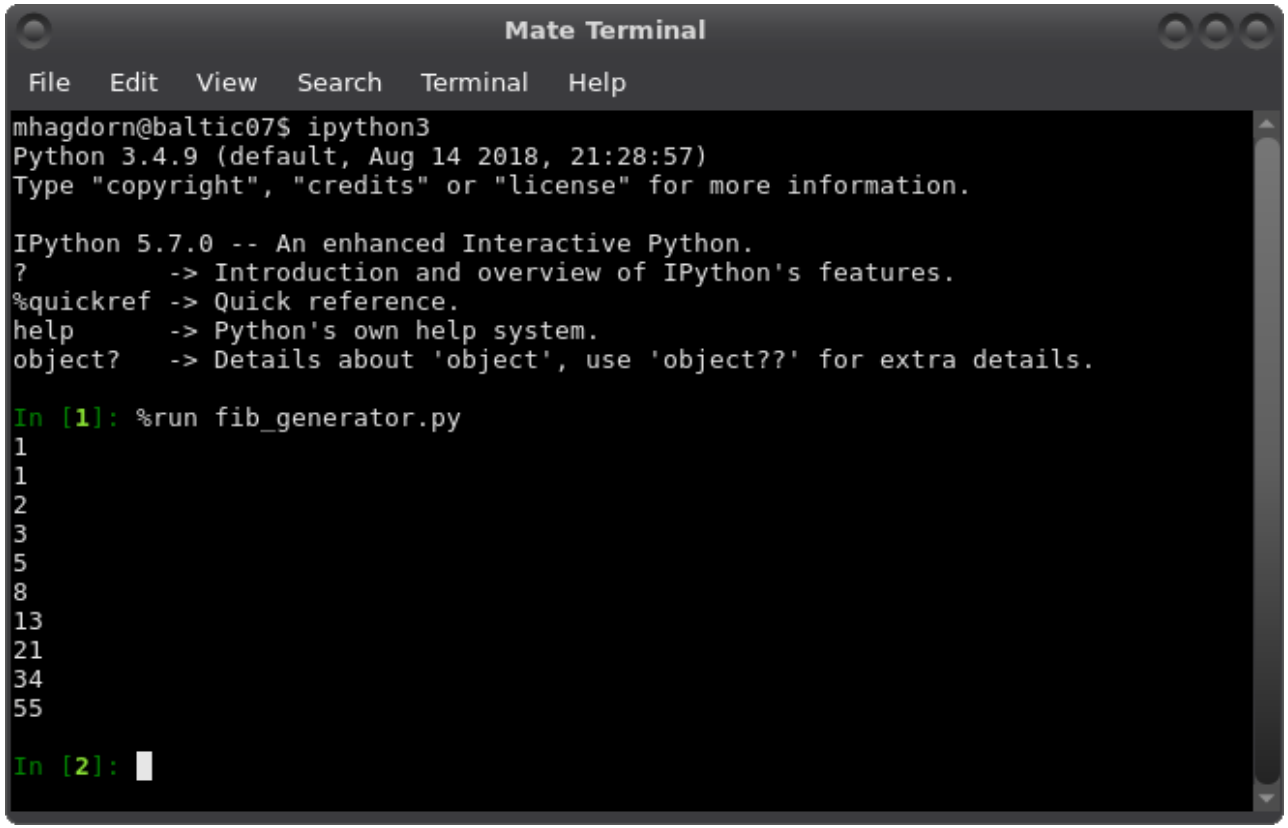
```
def fibg(k=None):
    a=1
    b=1
    while k is None or a<k:
        yield a
        a,b = b,a+b

if __name__ == '__main__':
    for f in fibg(60):
        print(f)
```

Debugging
interactive debugger

run programs from within ipython

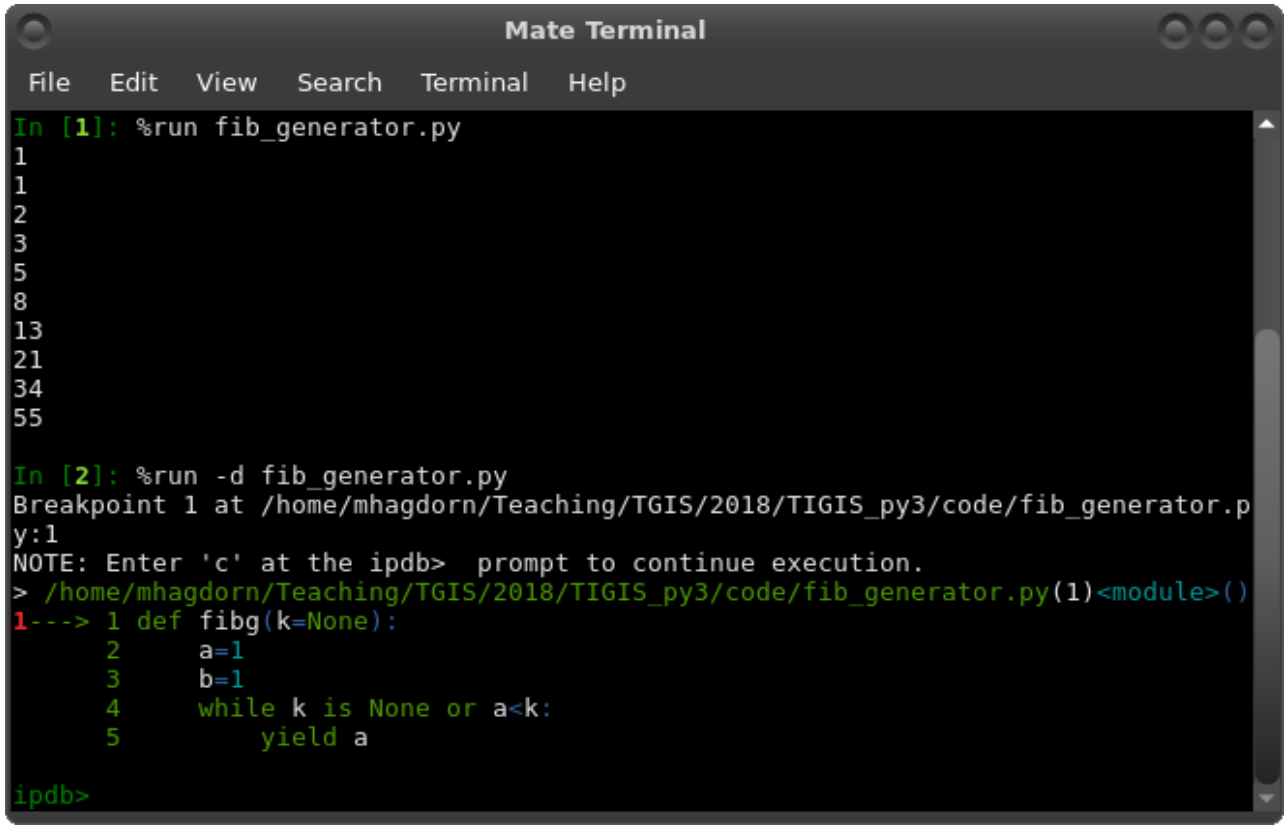
`%run pname.py`



Debugging
interactive debugger

debug programs using ipython

`%run -d pname.py`

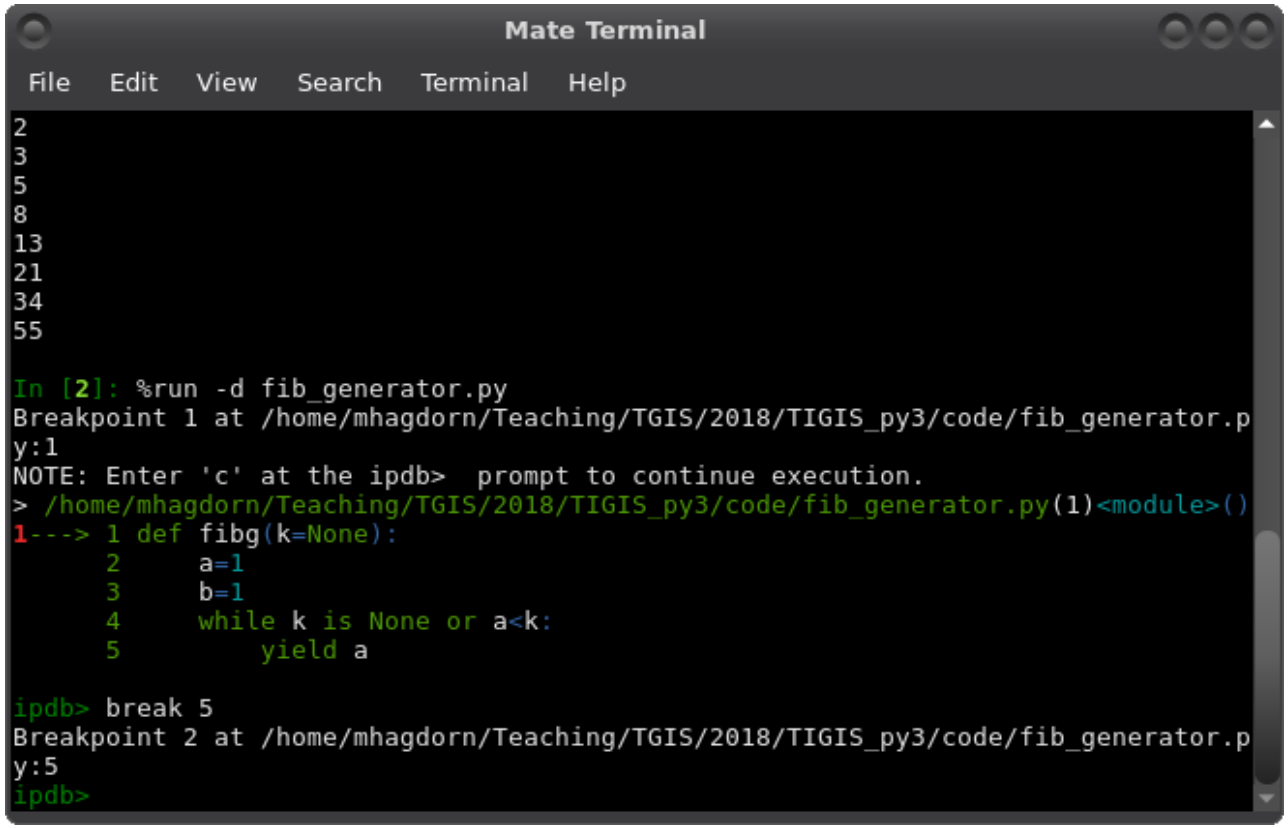


Debugging
debugger commands

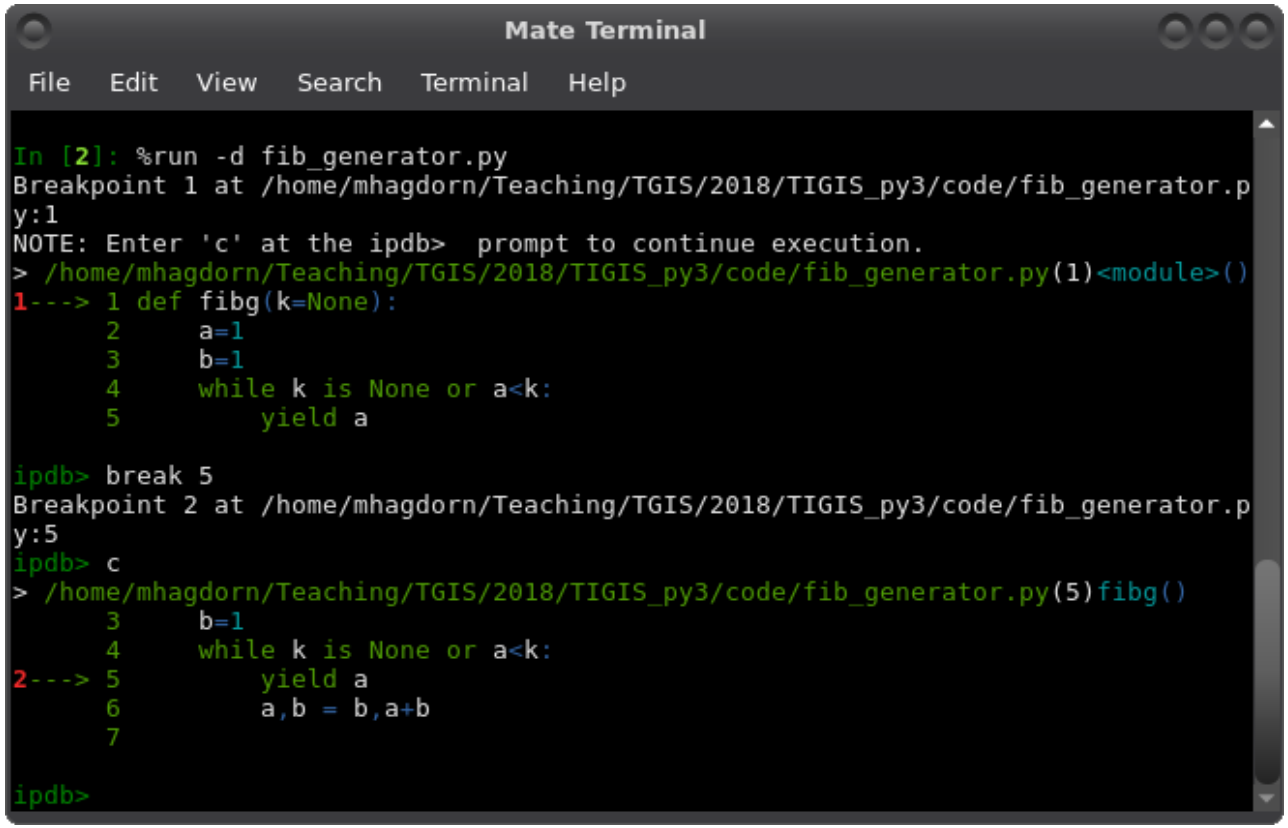
h(elp)	print help
n(ext)	execute the next line of code
s(tep)	execute the next line, if it is a function call stop inside the function
c(ontinue)	continue execution until next break point is reached
b(reak)	set break point
display	display value of expression each time it changes

many more commands, have a look at the pdb documentation

Debugging
debugging session

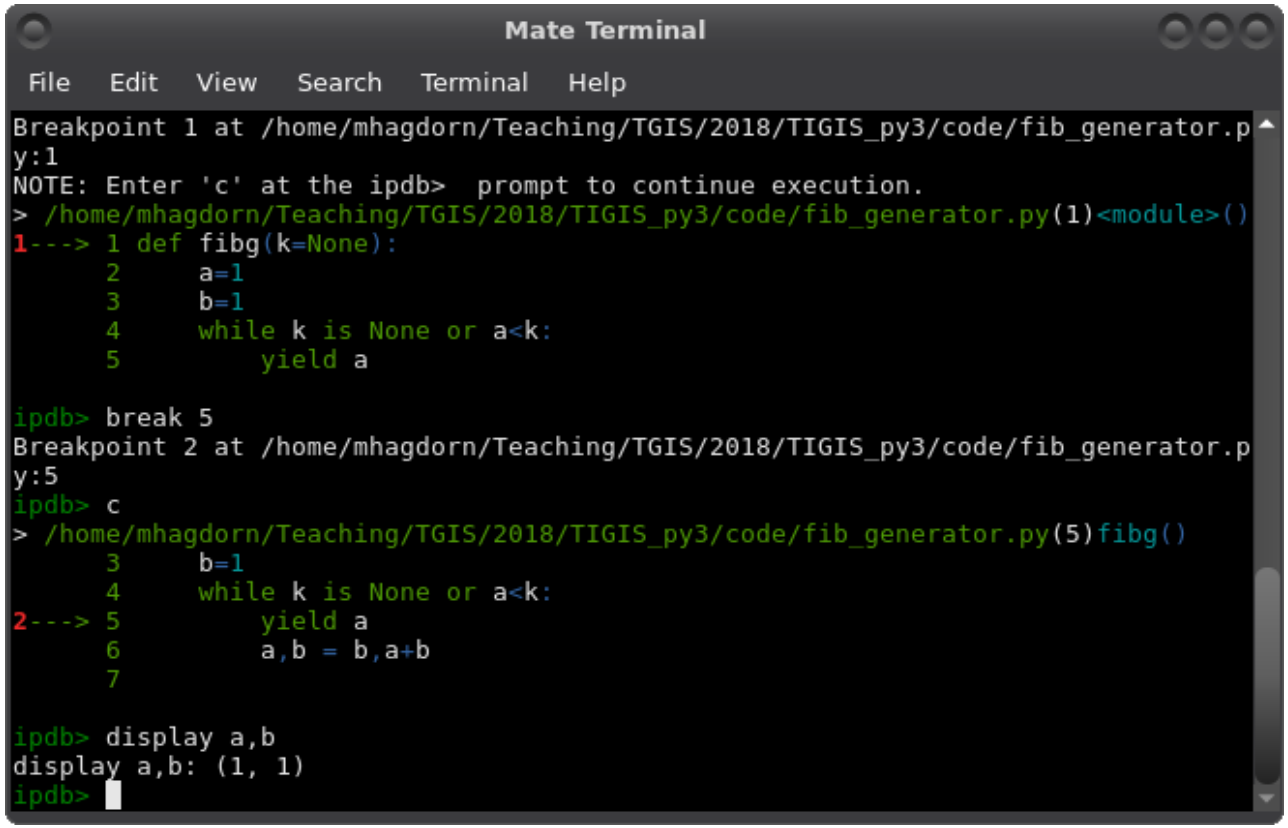


Debugging
debugging session

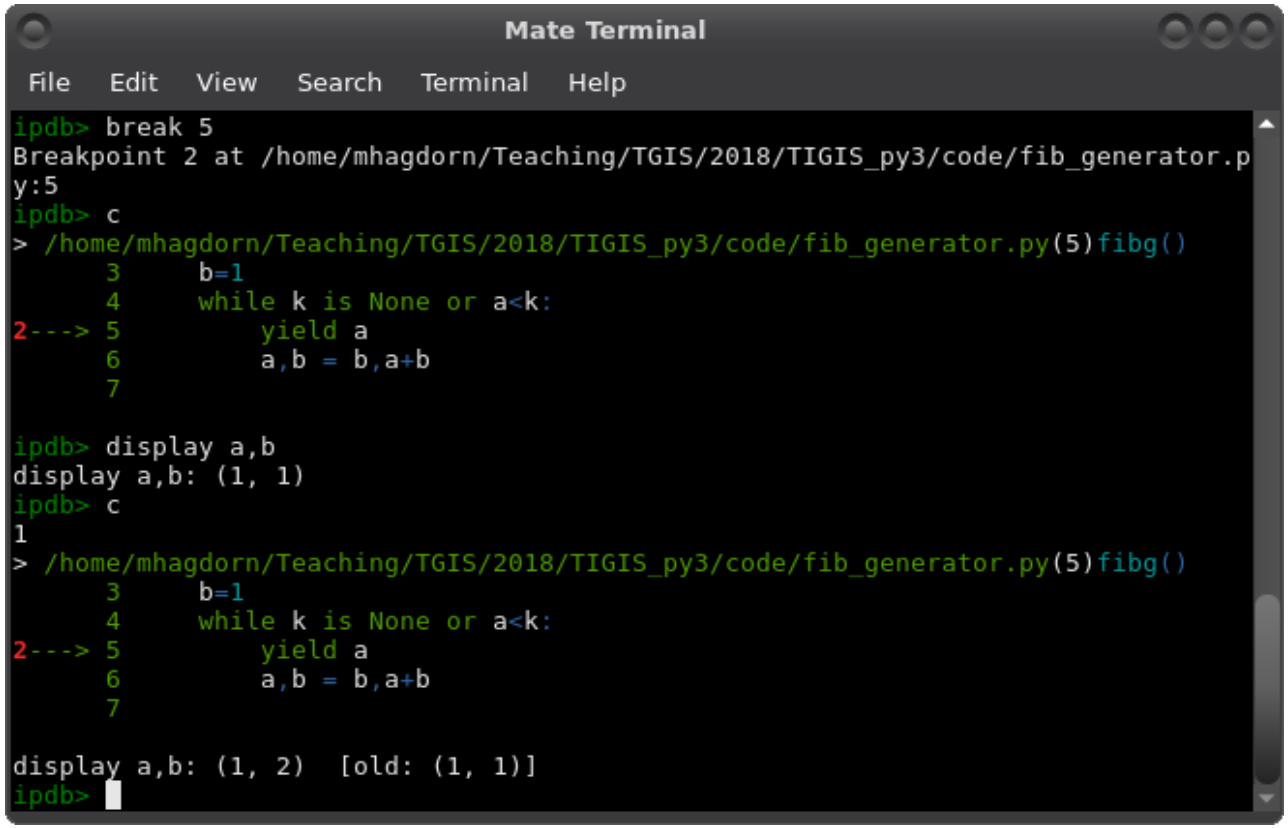


Debugging

debugging session



Debugging
debugging session



Debugging
instrumenting code


```
import ipdb

def fibg(k=None):
    a=1
    b=1
    while k is None or a<k:
        ipdb.set_trace()
        yield a
        a,b = b,a+b

if __name__ == '__main__':
    for f in fibg(60):
        print(f)
```

Debugging
alternatives

- use spyder - although that uses pdb under the hood
- use plenty of `print` statements
- use `pprint` from the pprint module

Debugging
Task

Exercise

- have a look at the two implementations of the Fibonacci numbers
- use the debugger in ipython to step through the code
- try out `pprint`

Testing

Making sure your code is (and keeps) working

Why testing

- allows you to be sure that your code works as expected
- allows you to change your code and still be sure that your code works as expected
- tests should be automatic
- make time to write tests

Different Kinds

Unit Testing

- smallest possible items are tested
- individual functions/methods
- usually done automatically using a test frame work

Integration Testing

The individual components that make up a system are tested together.

Testing
Something to test


```
def divide(a,b):
    """compute ratio of a and b"""

    return a/b
```

We are going to use `nose`

- create a directory for the tests: `mkdir tests`
- create a file `test_division.py` to test the `divide` function

```
from nose.tools import assert_equals
from division import divide
def test_division ():
    ratio = divide(1,2)
    assert ratio == 0.5
#    assert ratio == 0.5
    assert_equals ( ratio ,0.5)
```

Testing
Unit Testing

Run the test using

- `nosetests-3.6` to run all tests
- `nosetests-3.6 -v` to be more verbose
- `nosetests-3.6 test_division.py` to run tests of a particular file
- `nosetests-3.6 test_division.py:test_divsion` to run a particular function

Add another function to test what happens when you divide by zero:

```
from nose.tools import assert_equals, raises
@raises( ZeroDivisionError )
def test_divideByZero ():
    divide (1,0)
```

Sometimes you would like to test the same function for a number of values, you can use a generator for that:

```
def check_division(x, y, result):
    assert_equals (divide(x, y),result)

def test_many_div():
    for i in range(1,5):
        yield check_division, 2*i,i,2
        yield check_division, 2*i,str(i),2
```

coming up with test values is a pain. Let's generate them using the hypothesis module:

```
from hypothesis import given, assume
from hypothesis.strategies import integers

@given(x=integers())
def test_div_hypothesis(x):
    assume (x!=0)
    assert_equals (divide(2*x,x),2)
```

we can also test both floats and integers

```
from hypothesis import given,assume
from hypothesis.strategies import integers, floats
from hypothesis.strategies import one_of

@given(one_of(integers(), floats()))
def test_div_hypothesis2(x):
    assume (x!=0)
    assert_equals (divide(2*x,x),2)
```

Testing
Task

Exercise

- populate the `test_division.py` file
- Have a look at this introduction to nose.

Documentation

You should document your code!

[Public domain], via Wikimedia Commons

- others might want to use your code and need to know how
- after a while you will forget what your code does
- add a README, README.rst or README.md
  - restructured text
  - markdown text

Documentation

docstrings

You can automatically document your API using *docstrings* and the program pydoc/pydoc3 on the command line, eg

```
pydoc3 numpy
```

or

```
pydoc3 numpy.array
```

or

```
pydoc3 TIGISIO.load2Col
```

Documentation

docstrings

```
"""functions to load data files

"""

import numpy

__all__ = ['loadData2C']

def loadData2C(fname):
    """load data from a file with two columns

    Parameters
    ----------
    fname :
          the name of the file containing the data

    Example
    -------
    >>> x,y = loadData2C('some_file.data')

    """

    # create empty two lists which will hold the two columns of data
    x = []
    y = []
    # open the file
    inFile = open(fname,'r')
    # loop over each line of the file
    for line in inFile.readlines():
        # split the line and append to the data list
        # we assume that the columns are white-space separated
```

Documentation
docstrings

- there are various conventions for docstrings, have a look at the various docstring options. I quite like the numpy docstrings.
- you can use sphinx to generate HTML documentation
- have a look these instructions of how to setup sphinx for your project
- if you use github or bitbucket you can setup automatic doc builds on read the docs

Documentation

docstrings — tasks

Exercise

- annote your modules using docs strings
- try out pydoc
- give sphinx a go following the recipe https://samnicholls.net/2016/06/15/how-to-sphinx-readthedocs/