Numerical Python

CS*101* `lec14`

Errors & Exceptions

2018-11-19

Announcements

quiz: `quiz14` due on Tues 05/11

lab: `lab` on Fri 08/11

hw: `hw07` due 06/11

exam: `exam02` from `lec06-13` on `13 Nov` @ `8 pm`

Paper Exam - MCQs + Coding Qestions

Roadmap

Week 1 (9/9) L0 Intro L1 PY - Intro	Week 2 (16/9) L2 PY – Data& Bool L3 PY – if & Loops	Week 3 (23/9) L4 PY - Lists L5 PY - Funcs	30/9 L6 PY - Files ✗
9/10 L7 PY - Muta 12/10 Exam 01	Week 6 (14/10) L8 PY - Dict L9 PY - IO	Week 7 (21/10) L10 PY - Numpy L11 PY - Plot	Week 8 (28/10) L12 PY - State L13 PY - Rand
Week 9 (4/11) L14 PY – De-bug L15 PY - Eqn	Week 10 (11/11) L16 PYA I Exam 02	Week 11 (18/11) L17 PY - Opt II L18 PY - Mod	Week 12 (25/11) L19 PY – Sym I L20 PY – Sym II
Week 13 (2/12) L21 PY – Sym III L22 ML - Intro	Week 14 (9/12) L23 ML - IO L24 ML - Poly	Week 15 (16/12) L25 ML - Stats L26 ML - Fit	Week 16 (23/12) Review
Week 17 (30/12) Finals			

Objectives

Error, Errors Everywhere!

A. Explain the difference between errors, exceptions, and bugs.
B. Identify the types of exceptions raised by Python and their sources.
C. Use exception handling to avoid program crashes.
D. Identify why numerical (truncation) error occurs and when it is likely to do so, including countermeasures like ‘isclose‘ and ‘allclose‘.

State review

A. All models are wrong but some are more useful!

B. Each state tells the condition of the simulation at that particular time

C. Print() and Plot() to see what is happening

When Things Go Wrong...

After 6 hrs still wrong...

Common errors and exceptions

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ’:’, ’()’ etc. Displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little 'arrow' pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found

  B `TypeError` - operations on different data types

Common errors and exceptions

- A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little 'arrow' pointing at the earliest point in the line where the error was detected
- B `Exceptions` - errors found during running
  - A `NameError` - name of function or variables not found
  - B `TypeError` - operations on different data types
  - C `ZeroDivisionError`

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found

  B `TypeError` - operations on different data types

  C `ZeroDivisionError`

  D `FileNotFoundError`

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found

  B `TypeError` - operations on different data types

  C `ZeroDivisionError`

  D `FileNotFoundError`

  E `IndexError` - wrong index used

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found
  B `TypeError` - operations on different data types
  C `ZeroDivisionError`
  D `FileNotFoundError`
  E `IndexError` - wrong index used
  F `KeyError` - in dictionary

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little 'arrow' pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

  A `NameError` - name of function or variables not found
  B `TypeError` - operations on different data types
  C `ZeroDivisionError`
  D `FileNotFoundError`
  E `IndexError` - wrong index used
  F `KeyError` - in dictionary
  G `IndentationError` - wrong or no indent

Common errors and exceptions

A `SyntaxError` - also parsing error. Python does not like your structure. Missing ':', '()' etc. Displays a little 'arrow' pointing at the earliest point in the line where the error was detected

B `Exceptions` - errors found during running

   A `NameError` - name of function or variables not found
   B `TypeError` - operations on different data types
   C `ZeroDivisionError`
   D `FileNotFoundError`
   E `IndexError` - wrong index used
   F `KeyError` - in dictionary
   G `IndentationError` - wrong or no indent
   H .... more

Types of Bugs

A few working definitions:

- **Errors**—errors which cause the program to be unrunnable (cannot be handled at run time). In Python, it is mostly Syntax errors.

Types of Bugs

A few working definitions:

Errors—errors which cause the program to be unrunnable (cannot be handled at run time). In Python, it is mostly Syntax errors.

Exceptions—unusual behavior duing running (although not necessarily unexpected behavior, particularly in Python). Means there is no Syntax Error. Python liked your structure, run your code but found non-logic errors.

Types of Bugs

A few working definitions:

Errors—errors which cause the program to be unrunnable (cannot be handled at run time). In Python, it is mostly Syntax errors.

Exceptions—unusual behavior duing running (although not necessarily unexpected behavior, particularly in Python). Means there is no Syntax Error. Python liked your structure, run your code but found non-logic errors.

Bugs—include errors and exceptions, but also miswritten, ambiguous, or incorrect code which in fact runs but does not advertise its miscreancy (i.e., does not tell you that anything is wrong)

Common exceptions

A. `SyntaxError`—check missing colons or parentheses
B. `NameError`—check for typos, function definitions
C. `TypeError`—check variable types
D. `ValueError`—check function parameters
E. `FileNotFoundError`—check that files exist

Common exceptions

A. `IndexError`—don’t reference nonexistent list elements

B. `KeyError`—similar to an IndexError, but for dictionaries

C. `ZeroDivisionError`

D. `IndentationError`—check that spaces and tabs aren’t mixed

Question

```
# calculate squares
d = list(range(10))
while i < 10:
    d[i] = d[i] ** 2.0
    i += 1
```

Which error would this code produce?

- A `SyntaxError`
- B `IndexError`
- C `ValueError`
- D `NameError`

Question

```
# calculate squares
d = list(range(10))
while i < 10:
    d[i] = d[i] ** 2.0
    i += 1
```

Which error would this code produce?

- A `SyntaxError`
- B `IndexError`
- C `ValueError`
- D `NameError` ⋆

Question

Which of the following would produce `TypeError`?

- A `'2' + 2`
- B `2 / 0`
- C `2e8 + (1+0j)`
- D `'2' * 2`

Question

Which of the following would produce `TypeError`?

- A `'2' + 2` ⋆
- B `2 / 0`
- C `2e8 + (1+0j)`
- D `'2' * 2`

Program stack

Traceback—listing of function calls on the stack at the time the exception arises

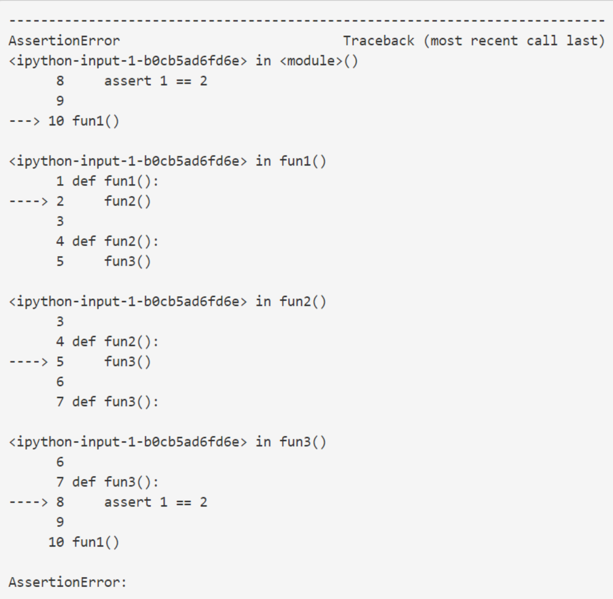
Program stack

Traceback—listing of function calls on the stack at the time the exception arises

```
def fun1():
    fun2()

def fun2():
    fun3()

def fun3():
    assert 1 == 2

fun1()
```

Program stack

```
fun1()
```



```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-1-b0cb5ad6fd6e> in <module>()
      8     assert 1 == 2
      9
---> 10 fun1()

<ipython-input-1-b0cb5ad6fd6e> in fun1()
      1 def fun1():
----> 2     fun2()
      3
      4 def fun2():
      5     fun3()

<ipython-input-1-b0cb5ad6fd6e> in fun2()
      3
      4 def fun2():
----> 5     fun3()
      6
      7 def fun3():

<ipython-input-1-b0cb5ad6fd6e> in fun3()
      6
      7 def fun3():
----> 8     assert 1 == 2
      9
     10 fun1()

AssertionError:
```

Handling Exceptions

Exception handling - Try

Most of the time, we do not want errors to happen.

Exception handling - Try

Most of the time, we do not want errors to happen.

But it happens

Exception handling - Try

Most of the time, we do not want errors to happen.

But it happens

Next best thing is we do not want our program to crash (stop executing)!

Exception handling - Try

Most of the time, we do not want errors to happen.

But it happens

Next best thing is we do not want our program to crash (stop executing)!

We can tell Python to `try` a block of code, and it will run normally `except` if something goes wrong.

Exception handling - Try

Most of the time, we do not want errors to happen.

But it happens

Next best thing is we do not want our program to crash (stop executing)!

We can tell Python to `try` a block of code, and it will run normally `except` if something goes wrong.

```
# calculate square roots
d = list( range( 10 ) )
r = []
for i in d:
    try:
        r[ i ] = sqrt( d[ i ] )
    except:
        print( ’An error occurred.’ )
        break
```

Try

The advantage: you can handle the error and **execution can proceed normally**.

The disadvantage: the traceback doesn’t appear automatically.

Try

The advantage: you can handle the error and **execution can proceed normally**.

The disadvantage: the traceback doesn’t appear automatically.

This also doesn’t guard against errors or bugs which don’t raise an exception: like your logic errors

Try structure

```
try:
    # the main code
    # if an error occurs, it goes into ”except:”
                             immediately
except:
    # an error occurs
else: (optional)
    # if no error occurs
finally: (optional)
    # this always happens, error or no error
```

Note: `except:` or `except XXXError:` both will work.
`XXXError` is the list of errors/exceptions from Python

Question

```
denom = 0
while True:
    try:
        # Read int from console/prompt.
        denom = input()

        # Use as denominator.
        i = 1 / float(denom)
        print(i)
    except:
        print(”non-numeric value entered”)
    else:
        print(i, ”again”)
    finally:
        if denom == ’q’: break
```

Examples

If we lose the information on what and where went wrong, our debugging step may not be appropriate.

Examples

If we lose the information on what and where went wrong, our debugging step may not be appropriate.

What could have gone wrong in the code below?

```
try:
    filename = 'spring.data'
    datafile = open( filename,'r' )
    data = datafile.readlines()
except:
    print( 'Something went wrong.' )
```

Be specific!!!

Use `try` at the finest degree of precision you can:

```
filename = ’spring.data’
try:
    datafile = open( filename,’r’ )
except:
    print( ’Unable to open file ”%s”.’%filename )
```

Be specific!!!

Use `try` at the finest degree of precision you can:

```
filename = 'spring.data'
try:
    datafile = open( filename,'r' )
except:
    print( 'Unable to open file "%s".'%filename )
```

is better than

```
filename = 'spring.data'
try:
    datafile = open( filename,'r' )
    for line in data:
        ...
except:
    ...
```

Question 1

```
a = [ 'a','n','y' ]
try:
    a[ 3 ] = '.'
except IndexError:
    pass  # does nothing
a[0][0] = 'b'
```

Which uncaught error will cause this code to terminate?

- A `IndexError`
- B `TypeError`
- C `KeyError`

Question 1

```
a = [ 'a','n','y' ]
try:
    a[ 3 ] = '.'
except IndexError:
    pass  # does nothing
a[0][0] = 'b'
```

Which uncaught error will cause this code to terminate?

A `IndexError`

B `TypeError` ⋆(where?)

C `KeyError`

Question 2

```
???
try:
    a[ 4 ] *= 2
except TypeError:
    pass
else:
    print( ’No error arose.’ )
```

Which line replacing the `???` will raise an uncaught error?

- A `a = ’12345’`
- B `a = [ 1,2,3,4 ]`
- C `a = ( 1,2,3,4,5 )`
- D `a = np.ones( ( 10, ) )`

Question 2

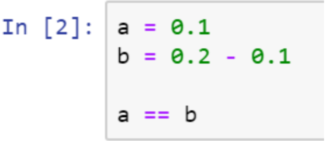
```
???
try:
    a[ 4 ] *= 2
except TypeError:
    pass
finally:
    print( 'No error arose.' )
```

Which line replacing the `???` will raise an uncaught error?

- A `a = '12345'`
- B `a = [ 1,2,3,4 ]` ⋆(why?)
- C `a = ( 1,2,3,4,5 )`
- D `a = np.ones( ( 10, ) )`

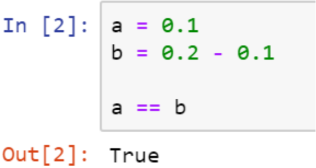
Numerical Error ???

How to compare floats?



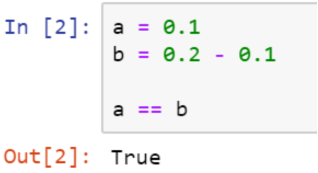
```
a = 0.1
b = 0.2 - 0.1

a == b
```

How to compare floats?

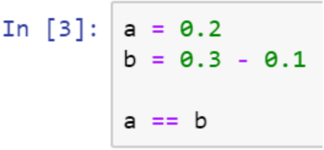


```
In [2]: a = 0.1
        b = 0.2 - 0.1

        a == b

Out[2]: True
```

How to compare floats?

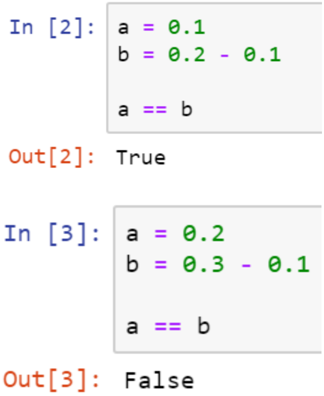


```
In [2]: a = 0.1
        b = 0.2 - 0.1

        a == b

Out[2]: True
```



```
In [3]: a = 0.2
        b = 0.3 - 0.1

        a == b
```

How to compare floats?



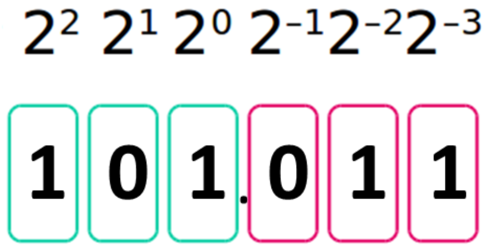
```
In [2]: a = 0.1
        b = 0.2 - 0.1

        a == b
Out[2]: True

In [3]: a = 0.2
        b = 0.3 - 0.1

        a == b
Out[3]: False
```

floats in binary

floats in binary

Represent *5.375* in binary?

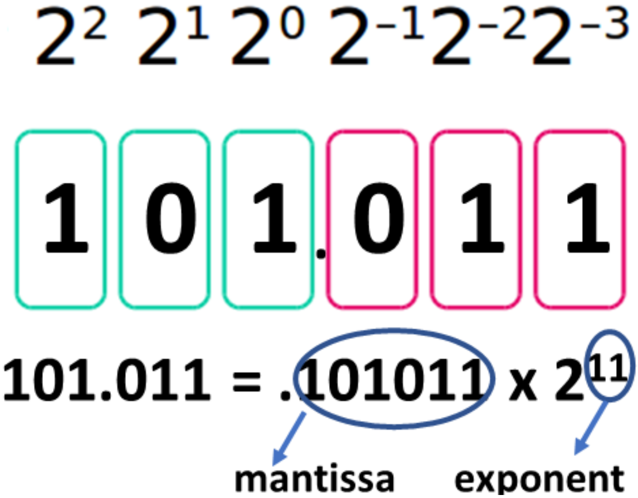
$$5.375 = 2^2 + 2^0 + 2^{-2} + 2^{-3}$$



$$2^2 \; 2^1 \; 2^0 \; 2^{-1} \; 2^{-2} \; 2^{-3}$$

$$1 \; 0 \; 1 . 0 \; 1 \; 1$$

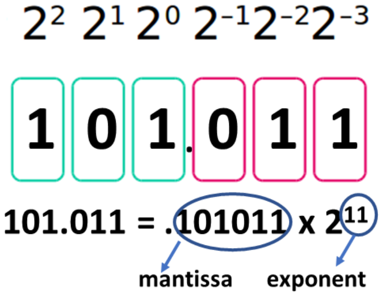
floats in binary?

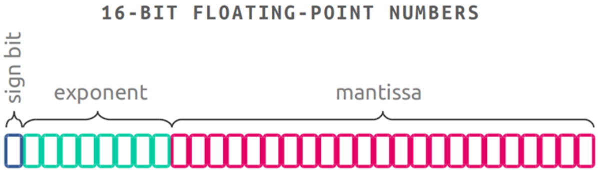


2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
1	0	1.	0	1	1

$$101.011 = .\underbrace{101011}_{\text{mantissa}} \times 2^{\overset{\text{exponent}}{11}}$$

floats in binary?





How to compare floats?

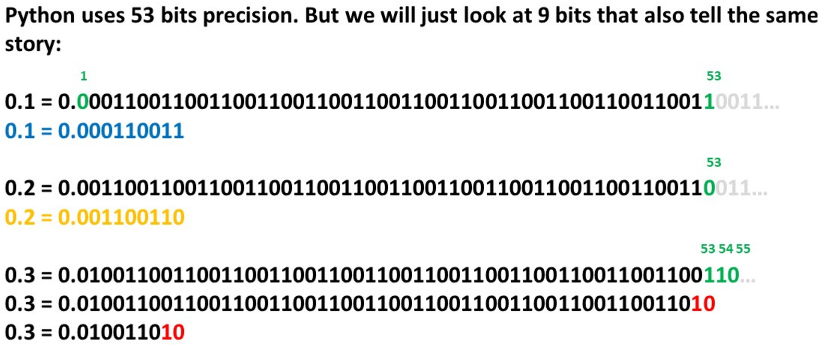
Python uses 53 bits precision. But we will just look at 9 bits that also tell the same story:

1 53

0.1 = 0.000110011001100110011001100110011001100110011001100110011...

0.1 = 0.000110011

How to compare floats?



Python uses 53 bits precision. But we will just look at 9 bits that also tell the same story:

1 53

0.1 = 0.0001100110011001100110011001100110011001100110011001100110011...

0.1 = 0.000110011

53

0.2 = 0.0011001100110011001100110011001100110011001100110011000 11...

0.2 = 0.001100110

53 54 55

0.3 = 0.0100110011001100110011001100110011001100110011001100110...

0.3 = 0.01001100110011001100110011001100110011001100110011010

0.3 = 0.010011010

How to compare floats?

Python uses 53 bits precision. But we will just look at 9 bits that also tell the same story:

1 53

0.1 = 0.000110011001100110011001100110011001100110011001100110011...

0.1 = 0.000110011

53

0.2 = 0.001100110011001100110011001100110011001100110011001100011...

0.2 = 0.001100110

53 54 55

0.3 = 0.0100110011001100110011001100110011001100110011001100110...

0.3 = 0.01001100110011001100110011001100110011001100110011010

0.3 = 0.010011010

0.2 - 0.1 = 0.000110011 same as 0.1

0.3 - 0.1 = 0.001100111 different from 0.2

How to compare floats?

from **math** or **numpy**,

```
math.isclose(a, b, rel_tol=1e-05, abs_tol=1e-08)
np.isclose( a, b, rtol=1e-05, atol=1e-08)
np.allclose(a, b, rtol=1e-05, atol=1e-08)
```

`rtol` = relative tolerance => abs(a - b) / abs(b)

`atol` = absolute rolerance => abs(a - b)

> `math.isclose()` compares numbers

> `np.isclose()` compares numbers or individual numbers in an array or list or tuple. It returns an array of bool if it compares an array or list or tuple.

> `np.allclose()` same as `np.isclose()` but returns only a single bool. Any `false` result from the array makes `np.allclose()` to return `false`

Debugging

Debugging

”Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Debugging

”Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

”The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”

Debugging

”Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

”The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”

”Controlling complexity is the essence of computer programming.”
— Brian Kernighan

Debugging strategies

When do things go wrong?

Debugging strategies

When do things go wrong?

Three categories of problems:

A. before the code runs
B. while the code is running
C. in the results

Debugging strategies

A. Start early.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.
E. Break the program down into functions.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.
E. Break the program down into functions.
F. Document functions before writing them.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.
E. Break the program down into functions.
F. Document functions before writing them.
G. Explain it to someone else.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.
E. Break the program down into functions.
F. Document functions before writing them.
G. Explain it to someone else.
H. Make no assumptions! If your thinking is not precise, your code will not be precise.

Debugging strategies

A. Start early.
B. Read the problem statement carefully.
C. Chart the flow of the program.
D. Add print statements.
E. Break the program down into functions.
F. Document functions before writing them.
G. Explain it to someone else.
H. Make no assumptions! If your thinking is not precise, your code will not be precise.
I. Start over from scratch. Take a fresh look at the problem.

Style

Style

Document your code!

Every function should have a docstring.

```
def warning( msg ):
    '''Display a warning message.'''
    print( 'Warning:  %s'%msg)
```

Style

Document your code!

Every function should have a docstring.

```
def warning( msg ):
    ’’’Display a warning message.’’’
    print( ’Warning:  %s’%msg)
```

Docstrings explain what the function does and what its parameters are.

They always are triple-quoted strings on the first line of the function block.

Style

Document your code!

Every function should have a docstring.

```
def warning( msg ):
    '''Display a warning message.'''
    print( 'Warning:  %s'%msg)
```

Docstrings explain what the function does and what its parameters are.

They always are triple-quoted strings on the first line of the function block.

```
help(warning)
```

Style

Use descriptive variable names.

Style

Use descriptive variable names.

Why do we write comments?

Style

Use descriptive variable names.

Why do we write comments?

For the person who next looks at the code! Also for YOU after you wrote the code a long time ago!

```
x_vals = [0,0.1,0.2,0.3,0.4]  # meters
faraday = 96485.3328959  # coulombs,
                         # electric charge
```

Style

Use functions to structure code.

This makes code more readable (and debuggable!).

Summary

A) List of different errors and exceptions (`NameError`, `SyntaxError` etc)

B) The source of these errors and how we can avoid them

C) Use `try` structure - at the finest degree of precision that you can

D) The source of numerical error for `float`