Data Science and Advanced Programming — Lecture 6a

Python Fundamentals III

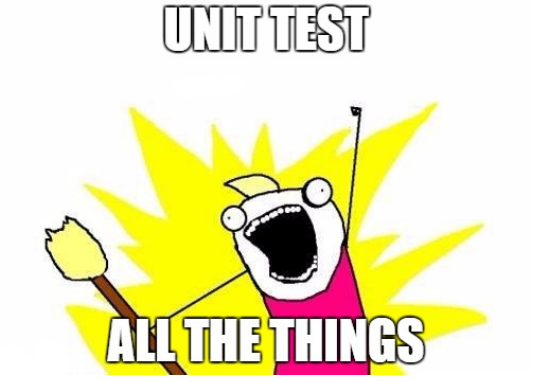
Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

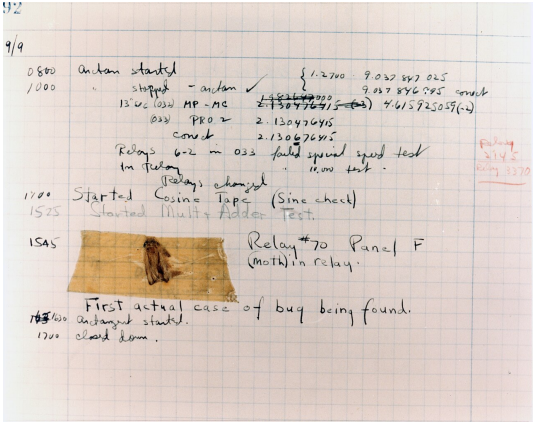
October 20th, 2025 | 12:30 - 16:00 | Internef 263

Roadmap

1. Basics on Testing and Debugging
2. The Python debugger — A debugging example (take-home)
3. Object Oriented Programming

1. Basics on Testing and Debugging





https://en.wikipedia.org/wiki/Software_bug

This class — so far

https://www.youtube.com/watch?v=1WF-HHDwa_Y

Reality of Coding

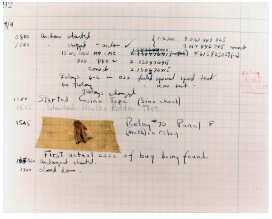
https://www.youtube.com/watch?v=v2qFM_Qsxgo

Mark II Aiken Relay Computer (1947)

https://en.wikipedia.org/wiki/Debugging

Mark II, general view of calculator frontpiece, 1948.

Admiral Grace Hopper



Example — a buggy code

demo/example_25.py

```
########################################
## EXAMPLE: Buggy code to reverse a list
########################################
def rev_list_buggy(L):
    """
    input: L, a list
    Modifies L such that its elements are in reverse order
    returns: nothing
    """
    for i in range(len(L)):
        j = len(L) - i
        L[i] = temp
        L[i] = L[j]
        L[j] = L[i]
# FIXES: -------------------------
# temp unknown
# list index out of range -> sub 1 to j
# get same list back -> iterate only over half
# -------------------------
L = [1,2,3,4]
rev_list_buggy(L)  #call buggy code
print(L)
```

Example — a debugged code

demo/example_25.py

```
### debugged code
def rev_list(L):
    """
    input: L, a list
    Modifies L such that its elements are in reverse order
    returns: nothing
    """
    for i in range(len(L)//2):
        j = len(L) - i - 1
        temp = L[i]
        L[i] = L[j]
        L[j] = temp
```

Ensure functionality of the code

Defensive Coding

- Write specifications for functions
- Modularize programs
- Check conditions on inputs/outputs (assertions)

- TESTING/VALIDATION
  - Compare input/output pairs to specification
  - "It's not working!"
  - "How can I break my program?"
- DEBUGGING
  - Study events leading up to an error
  - "Why is it not working?"
  - "How can I fix my program?"

Set yourself up for testing and debugging

- from the start, design code to ease this part
- **break program up into modules that can be tested and debugged individually**
- document constraints on modules
  - what do you expect the *input* to be?
  - what do you expect the *output* to be?
- document assumptions behind code design

When can you start to test the functionality?

- **ensure code runs**
  - remove syntax errors
  - remove static semantic errors
  - **Python interpreter can usually find these for you**
- **have a set of expected results**
  - an input set
  - for each **input**, the expected **output**

3 Types of Testing — a cycle back

1. Unit testing
   - validate each piece of program
   - testing each function separately
2. Regression testing
   - add test for bugs as you find them
   - catch re-introduced errors that were previously fixed
3. Integration testing
   - does overall program work?
   - tend to rush to do this

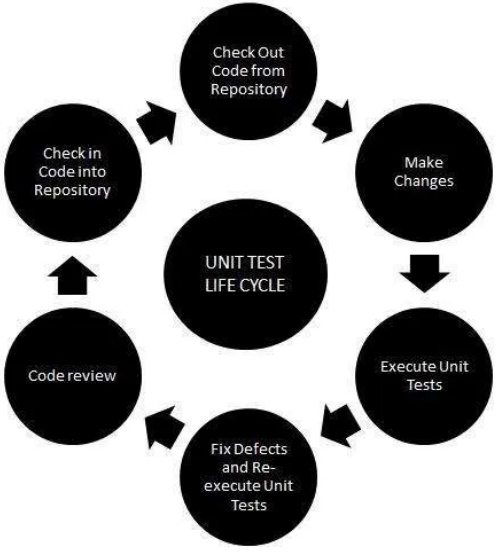
What is Unit Testing?

- a testing technique using which individual modules are tested to determine if there are any issues by the developer himself.
- It is concerned with functional correctness of the standalone modules.
- The main aim is toisolate each unit of the system to identify, analyse and fix the defects.

Unit Testing — Advantages

- Reduces defects in the newly developed features or reduces bugs when changing the existing functionality.
- Reduces cost of testing as defects are captured in very early phase.
- Improves design and allows better re-factoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well

Unit Testing — life cycle (cf. Git)



Black Box Testing (assume the tester has the docstring)

- designed without looking at the code
- can be done by someone other than the implementer to avoid some implementer biases
- testing can be reused if implementation changes
- paths through specification
  - build test cases in different natural space partitions
  - also consider boundary conditions (empty lists, singleton list, large numbers, small numbers)

Black Box testing — Example

```
def sqrt(x, eps):
""" Assumes x, eps floats, x >= 0, eps > 0
Returns res such that x-eps <= res*res <= x+eps """
```

Figure out test cases without looking at the code!

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

Glass Box Testing (you have the code)

- use code directly to guide design of test cases
- called **path-complete** if every potentialpath through code is tested at least once
- what are some drawbacks of this type of testing?
  - can go through loops arbitrarily many times
  - missing paths
- guidelines
  - branches
  - for loops
  - while loops

The Purpose of Debugging

- The process of detecting and removing of existing and potential errors (also called as 'bugs') in a software code that can cause it to behave unexpectedly or a software crash.
- To prevent incorrect operation of a software or system, debugging is used to find and resolve bugs or defects.
- When various subsystems or modules are tightly coupled, debugging becomes harder as any change in one module may cause more bugs to appear in another.
- Sometimes it takes more time to debug a program than to code it.

High-level Debugging

- To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it.
- A user of a program must know how to fix the problem as knowledge about problem analysis is expected.
- When the bug is fixed, then the software is ready to use.
- Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables.

High-level Debugging

- To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it.
- A user of a program must know how to fix the problem as knowledge about problem analysis is expected.
- When the bug is fixed, then the software is ready to use.
- Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables.

High-level Debugging

- To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it.
- A user of a program must know how to fix the problem as knowledge about problem analysis is expected.
- When the bug is fixed, then the software is ready to use.
- Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables.

High-level Debugging

- To debug a program, user has to start with a problem, isolate the source code of the problem, and then fix it.
- A user of a program must know how to fix the problem as knowledge about problem analysis is expected.
- When the bug is fixed, then the software is ready to use.
- Debugging tools (called debuggers) are used to identify coding errors at various development stages. They are used to reproduce the conditions in which error has occurred, then examine the program state at that time and locate the cause. Programmers can trace the program execution step-by-step by evaluating the value of variables and stop the execution wherever required to get the value of variables or reset the program variables.

Typical steps in Debugging

- Reproduce the problem.
- Describe the bug. Try to get as much input from the user to get the exact reason.
- Capture the program snapshot when the bug appears. Try to get all the variable values and states of the program at that time.
- Analyze the snapshot based on the state and action. Based on that try to find the cause of the bug.
- **Fix the existing bug, but also check that any new bug does not occur.**

Debugging tools in Python

Tools:

- Pyflakes, pylint, PyChecker,...
- Python Tutor
- print statements
- **your brain**
- ...

`print` statements

- "print statements" are a good way to **test hypothesis**
- when to print?
  - enter function
  - parameters
  - function results

Tests triggering Error Messages

- trying to access beyond the limits of a list `test = [1,2,3]` then `test [4]` → `IndexError`
- trying to convert an inappropriate type `int(test)` → `TypeError`
- referencing a non-existent variable `a` → `NameError`
- mixing data types without appropriate coercion: `'3' / 4` → `TypeError`
- forgetting to close parenthesis, quotation, etc.
  `a=len([1,2,3]`
  `print(a)` → `SyntaxError`

Logical Errors — the hardest case

- Logical errors are the most difficult to fix.
- They occur when the program runs without crashing, but produces an incorrect result.
- The error is caused by a mistake in the program's logic.
- You won't get an error message, because no syntax or runtime error has occurred.
- You will have to find the problem on your own by reviewing all the relevant parts of your code - although some tools can flag suspicious code which looks like it could cause unexpected behavior.

Logical errors (II)

- Sometimes there can be absolutely nothing wrong with your Python implementation of an algorithm — the algorithm itself can be incorrect.
- However, more frequently these kinds of errors are caused by programmer carelessness.
- Here are some examples of mistakes which lead to logical errors:
  - using the wrong variable name
  - indenting a block to the wrong level
  - using integer division instead of floating-point division
  - getting operator precedence wrong
  - making a mistake in a boolean expression
  - Off-by-one, and other numerical errors
  - If you misspell an identifier name, you may get a runtime error or a logical error, depending on whether the misspelled name is defined.

Runtime Errors — Program Crash

- **If a program is syntactically correct — that is, free of syntax errors — it will be run by the Python interpreter.**
- However, the program may exit unexpectedly during execution if it encounters a runtime error - a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed.
- When a program comes to a halt because of a runtime error, we say that it **has crashed**.

Runtime Errors (II)

Some examples of Python runtime errors:

- division by zero.
- performing an operation on incompatible types.
- using an identifier which has not been defined.
- accessing a list element, dictionary value or object attribute which doesn't exist.
- trying to access a file which doesn't exist.

Runtime errors often creep in if you don't consider all possible values that a variable could contain, especially when you are processing user input. You should always try to add checks to your code to make sure that it can deal with bad input and edge cases gracefully.

Handling Exceptions I

- Until now, the programs that we have written have generally ignored the fact that things can go wrong.
- We have tried to prevent runtime errors by checking data which may be incorrect before we used it, but we haven't yet seen how we can handle errors when they do occur - our programs so far have just crashed suddenly whenever they have encountered one.
- There are some situations in which runtime errors are likely to occur.
- **Whenever we try to read a file or get input from a user, there is a chance that something unexpected will happen** - the file may have been moved or deleted, and the user may enter data which is not in the right format.
- Good programmers should add safeguards to their programs so that common situations like this can be handled gracefully - a program which crashes whenever it encounters an easily foreseeable problem is not very pleasant to use.
- **Most users expect programs to be robust enough** to recover from these kinds of setbacks.

Handling Exceptions II

- If we know that a particular section of our program is likely to cause an error, we can tell Python what to do if it does happen.
- Instead of letting the error crash our program we can intercept it, do something about it, and allow the program to continue.
- All the **runtime** (and syntax) errors that we have encountered are called **exceptions** in Python.
- Python uses them to indicate that something exceptional has occurred, and that your program cannot continue unless it is handled.
- All exceptions are sub-classes of the *Exception* class.

Exceptions

- what happens when procedure execution hits an unexpected condition?
- get an exception... to what was expected
- trying to access beyond list limits `test = [1,7,4]`
  `test[4]` → `IndexError`
- trying to convert an inappropriate type `int(test)` → `TypeError`
- referencing a non-existing variable a → `NameError`
- mixing data types without coercion `'a'/4` → `TypeError`

Exceptions (II)

Other common error types:

- `Syntaxerror`: Python can't parse program
- `NameError`: local or global name not found
- `AttributeError`: attribute reference fails
- `TypeError`: operand doesn't have correct type
- `ValueError`: operand type okay, but value is illegal
- `IOError`: IO system reports malfunction (e.g. file not found)

How to deal with Exceptions?

demo/example_26.py

- To handle possible exceptions, we use a `try-except` block:

```
try:
    age = int(input("Please enter your age: "))
    print("I see that you are %d years old." % age)
except ValueError:
    print("Hey, that wasn't a number!")
```

- Python will try to process all the statements inside the `try` block.
- If a `ValueError` occurs at any point as it is executing them, the flow of control will immediately pass to the except block, and any remaining statements in the try block will be skipped.
- In this example, we know that the error is likely to occur when we try to convert the user's input to an integer. If the input string is not a number, this line will trigger a `ValueError` — that is why we specified it as the type of error that we are going to handle.

Handling specific Exceptions

- We could have specified a more general type of error — or even left the type out entirely, which would have caused the except clause to match any kind of exception — but that would have been a bad idea.
- What if we got a completely different error that we hadn't predicted?
- It would be handled as well, and we wouldn't even notice that anything unusual was going wrong.
- We may also want to **react in different ways to different kinds of errors**. We should always try to pick specific rather than general error types for our except clauses.

Handling specific Exceptions (II)

demo/example_26b.py

- It is possible for one except clause to handle more than one kind of error: we can provide **a tuple of exception types** instead of a single type:

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except(ValueError, ZeroDivisionError):
    print("Oops, something went wrong!")
```

- A try-except block can also have multiple except clauses. If an exception occurs, Python will check each except clause from the top down to see if the exception type matches.

Handling specific Exceptions (III)

demo/example_26c.py

- If n**one of the except clauses match**, the exception will be considered un-handled, and **your program will crash:**

```
try:
    dividend = int(input("Please enter the dividend: "))
    divisor = int(input("Please enter the divisor: "))
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ValueError:
    print("The divisor and dividend have to be numbers!")
except ZeroDivisionError:
    print("The dividend may not be zero!")
```

- Note that in the example above if a ValueError occurs we won't know whether it was caused by the dividend or the divisor not being an integer — either one of the input lines could cause that error.

Handling specific Extensions IV

demo/example_26d.py

- If we want to give the user more specific feedback about which input was wrong, we will have to wrap each input line in a separate try-except block:

```
try:
    dividend = int(input("Please enter the dividend: "))
except ValueError:
    print("The dividend has to be a number!")
try:
    divisor = int(input("Please enter the divisor: "))
except ValueError:
    print("The divisor has to be a number!")
try:
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ZeroDivisionError:
    print("The dividend may not be zero!")
```

- In general, it is a better idea to use exception handlers to protect small blocks of code against specific errors than to wrap large blocks of code and write vague, generic error recovery code.
- It may sometimes seem inefficient and verbose to write many small try-except statements instead of a single catch-all statement, but we can mitigate this to some extent by making effective use of loops and functions to reduce the amount of code duplication.

Another Example

demo/example_27.py

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

The `else` and `finally` statements

`demo/example_28.py`

- There are two other clauses that we can add to a try-except block:

  `else` and **`finally`**

- `else` will be executed only if the try clause doesn't raise an exception:

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
```

The `else` and `finally` statements II

- We want to print a message about the user's age only if the integer conversion succeeds.
- In the first exception handler example, we put this print statement directly after the conversion inside the try block.
- In both cases, the statement will only be executed if the conversion statement doesn't raise an exception, but putting it in the `else` block is better practice — it means that the only code inside the try block is the single line that is the potential source of the error that we want to handle.

The `else` and `finally` statements III

- When we edit this program in the future, we may introduce additional statements that should also be executed if the age input is successfully converted.
- Some of these statements may also potentially raise a `ValueError`.
- If we don't notice this, and put them inside the try clause, the except clause will also handle these errors if they occur.
- This is likely to cause some odd and unexpected behaviour.
- By putting all this extra code in the else clause instead, we avoid taking this risk.

The `else` and `finally` statements IV

`demo/example_29.py`

- The `finally` clause will be executed at the end of the try-except block **no matter what** — if there is no exception, if an exception is raised and handled, if an exception is raised and not handled, and even if we exit the block using break, continue or return.
- **We can use the `finally` clause for cleanup code that we always want to be executed:**

```
try:
    age = int(input("Please enter your age: "))
except ValueError:
    print("Hey, that wasn't a number!")
else:
    print("I see that you are %d years old." % age)
finally:
    print("It was really nice talking to you.  Goodbye!")
```

The `with` statement

- Python's exception objects contain more information than just the error type. They also come with some kind of message
- Often these messages aren't very user-friendly — if we want to report an error to the user we usually need to write a more descriptive message which explains how the error is related to what the user did.
- For example, if the error was caused by incorrect input, it is helpful to tell the user which of the input values was incorrect.

The with statement II

demo/example_30.py

- Sometimes the exception message contains useful information which we want to display to the user.
- In order to access the message, we need to be able to access the exception object. We can assign the object to a variable that we can use inside the except clause like this:

```
try:
    age = int(input("Please enter your age: "))
except ValueError as err:
    print(err)
```

The `with` statement III

demo/example_31.py

- `err` is not a string, but Python knows how to convert it into one — the string representation of an exception is the message, which is exactly what we want.
- We can also combine the exception message with our own message:

```
try:
    age = int(input("Please enter your age: "))
except ValueError as err:
    print("You entered incorrect age input: %s" % err)
```

- Note that inserting a variable into a formatted string using %s also converts the variable to a string.

Raising exceptions

demo/example_32.py

- We can **raise exceptions** ourselves using the raise statement:

```
try:
    age = int(input("Please enter your age: "))
    if age < 0:
        raise ValueError("%d is not a valid age. Age must be positive or zero.")
except ValueError as err:
    print("You entered incorrect age input: %s" % err)
else:
    print("I see that you are %d years old." % age)
```

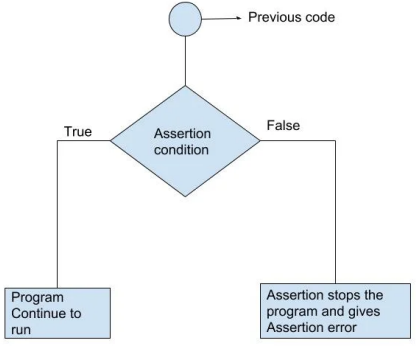
- We can raise our own `ValueError` if the age input is a valid integer, but it's negative.
- When we do this, it has exactly the same effect as any other exception — the flow of control will immediately exit the try clause at this point and pass to the except clause.
- This `except` clause can match our exception as well, since it is also a `ValueError`.

Raising exceptions

- We picked ValueError as our exception type because it's the most appropriate for this kind of error.
- There's nothing stopping us from using a completely inappropriate exception class here, but we should try to be consistent. Here are a few common exception types which we are likely to raise in our own code:
- `TypeError:` this is an error which indicates that a variable has the wrong type for some operation. We might raise it in a function if a parameter is not of a type that we know how to handle.
- `ValueError:` this error is used to indicate that a variable has the right type but the wrong value. For example, we used it when age was an integer, but the wrong kind of integer.
- `NotimplementedError:` exception used to indicate that a class's method has to be implemented in a child class (see later)

Assertions

- want to be sure that assumptions on state of computation are as expected.
- use an assert statement to raise an `AssertionError` exception if assumptions not met.
- This is an **example of good defensive programming.**



Assertions (II)

- Assertions don't allow a programmer to control response to unexpected conditions.
- ensure that **execution halts whenever an expected condition is not met.**
- typically used to check inputs to functions, but can be used anywhere.
- can be used to check outputs of a function to **avoid propagating** bad values.
- can make it easier to locate a source of a bug.

Using assert without Error Message

demo/example_34.py

- Syntax for using Assert in Python:

```
assert <condition>
assert <condition>,<error message>
```

```
def avg(marks):
    assert len(marks) != 0
    return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

- When we run the above program, the output will be: `AssertionError`
- We got an error as we passed an **empty list** `mark1` to assert statement, **the condition became false and assert stops the program** and give `AssertionError`.

Using assert with error message

demo/example_35.py

- Now let's pass another list which will satisfy the assert condition and see what will be our output

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

- When we run the above program, the output will be:

```
Average of mark2: 78.0
AssertionError: List is empty.
```

- We passed a non-empty list `mark2` and also an empty list `mark1` to the `avg()` function and we got output for `mark 2` list but after that we got an error `AssertionError: List is empty`.
- The assert condition was satisfied by the `mark2` list and program to continue to run.
- However, `mark1` doesn't satisfy the condition and gives an AssertionError.

Where and when to use Assertions

- goal is to spot bugs as soon as introduced and make clear where they happened.
- use as a supplement to testing.
- raise exceptions if users supplies bad data input
- use assertions to check types of arguments or values
  - check that invariants on data structures are met
  - check constraints on return value
  - check for violations of constraints on procedure (e.g. no duplicates in a list)

2. The Python debugger

Using a Debugger

- A debugger is a program that can help you to find out what is going on in a computer program.
- You can
  - stop the execution at any prescribed line number
  - print out variables
  - continue execution
  - stop again
  - execute
- statements one by one, and repeat such actions until you track down abnormal behavior and find bugs.

Demonstration of a debugger

`demo/Simpson.py`

- Example: Numerical Integration
- An integral $\int_a^b f(x)dx$
- can be approximated by the so-called Simpson's rule:

$$\int_a^b f(x)dx \approx \frac{b-a}{3n}\left(f(a)+f(b)+4\sum_{i=1}^{n/2} f(a+(2i-1)h)+2\sum_{i=1}^{n/2-1} f(a+2ih)\right).$$

- Here, $h=(b-a)/n$ and n must be an even integer.

$$\longrightarrow \frac{3}{2}\int_0^{\pi} \sin^3 x dx$$

Demonstration of a debugger — steps

demo/Simpson.py

- Start IPython: `In [1]: run -d Simpson.py`
- We now enter the debugger and get a prompt `ipdb>`
- After this prompt we can issue various debugger commands. The most important ones will be described as we go along.
- Type `continue` or just `c` to go to the first line in the file. Now you can see a printout of where we are in the program:

```
1---> 1 def Simpson(f, a, b, n=500):
      2     """
      3   Return the approximation of the integral of f
```

- Each program line is numbered and the arrow points to the next line to be executed. This is called the current line.

Demonstration of a debugger — steps

demo/Simpson.py

- You can set a break point where you want the program to stop so that you can examine variables and perhaps follow the execution closely. We start by setting a break point in the application function:

```
ipdb> break application
Breakpoint 2 at /home/.../src/funcif/Simpson.py:30
```

- You can also say break x, where X is a line number in the file.
- Continue execution until the break point by writing continue or c. Now the program stops at line 31 in the application function:

```
ipdb> c
> /home/.../src/funcif/Simpson.py(31)application()
2    30 def application():
---> 31     from math import sin, pi
     32     print("Integral of 1.5*sin^3 from 0 to pi:")
```

Demonstration of a debugger — steps

demo/Simpson.py

- Typing step or just s executes one statement at a time. Let us test this feature:

```
ipdb> s
> /home/.../src/funcif/Simpson.py(32)application()
     31     from math import sin, pi
---> 32     print("Integral of 1.5*sin^3 from 0 to pi:")
     33     for n in 2,6,12,100,500:

ipdb> s
Integral of 1.5*sin^3 from 0 to pi:
> /home/.../src/funcif/Simpson.py(33)application()
     32     print("Integral of 1.5*sin^3 from 0 to pi:")
---> 33     for n in 2,6,12,100,500:
     34         approx = Simpson(h, 0, pi, n)
```

- Typing another s reaches the call to Simpson, and a new s steps into the function Simpson:

```
ipdb> s
--Call--
> /home/.../src/funcif/Simpson.py(1)Simpson(1)
1---> 1 def Simpson(f, a, b, n=500):
      2     """
      3   Return the approximation of the integral of f
```

- Type a few more s to step ahead of the if tests.

Demonstration of a debugger — steps

demo/Simpson.py

- Examining the contents of variables is easy with the print (or p) command:

```
ipdb> print(f, a, b, n)
<function h at 0 x 898ef44> 0 3.14159265359 2
```

- We can also check the type of the objects by typing whatis ...

```
ipdb> whatis f
Function h
ipdb> whatis a
<type'int'>
ipdb> whatis b
<type'float'>
ipdb> whatis n
<type'int'>
```

- Set a new break point in the application function so that we can jump directly there without having to go manually through all the statements in the Simpson function.

Demonstration of a debugger — steps

demo/Simpson.py

- To see line numbers and corresponding statements around some line with number X, type list X. For example,

```
ipdb>list 32
27 def h(x):
28     return (3./2)*sin(x)**3
29
30 from math import sin, pi
31
32 def application():
33     print('Integral of 1.5*sin^3 from 0 to pi:')
34     for n in 2, 6, 12, 100, 500:
35         approx = Simpson(h, 0, pi, n)
36         print( 'n=%3d, approx=%18.15f, error=%9.2E' % \
37                (n, approx, 2-approx))
```

- We set a line break at line 35 :

```
ipdb>break 35
Breakpoint 3 at /home/.../src/funcif/Simpson.py:35
```

- Typing c continues execution up to the next break point, line 35.

Demonstration of a debugger — steps

demo/Simpson.py

- The command `next` or `n` is like step or `s` in that the current line is executed, but the execution does not step into functions, instead the function calls are just performed and the program stops at the next line

```
ipdb> n
> /home/.../src/funcif/Simpson.py(36)application()
3    35         approx = Simpson(h, 0, pi, n)
---> 36         print( 'n=%3d, approx=%18.15f, error=%9.2E' % \
     37                (n, approx, 2-approx))
ipdb> print approx, n
1.9891717005835792 6
```

- The command `disable X Y Z` disables break points with numbers X, Y, and Z, and so on. To remove our three break points and continue execution until the program naturally stops, we write

```
ipdb> disable 1 2 3
ipdb> c
n=100, approx=1.999999902476350, error=9.75E-08
n=500, approx=1.999999999844138, error=1.56E-10

In [2]:
```

a debugger is a very handy tool for monitoring the program flow, checking variables, and thereby understanding why errors occur.

Questions?