COMPSCI 590N
Lecture 11: Debugging and Exceptions

Roy J. Adams

College of Information and Computer Sciences
University of Massachusetts Amherst

Outline

Debugging

- **Debugging** is the process of finding the errors in your code.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.
- Python has a number of these:

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.
- Python has a number of these:
  - **pdb** - the built-in debugger.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.
- Python has a number of these:
  - **pdb** - the built-in debugger.
  - **DDD** - graphical front-end for many debuggers including pdb.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.
- Python has a number of these:
  - **pdb** - the built-in debugger.
  - **DDD** - graphical front-end for many debuggers including pdb.
  - **Spyder** - Spyder has a built in debugger.

Debugging

- **Debugging** is the process of finding the errors in your code.
- This process can take as much or more time as writing the code itself.
- Having skill at debugging can save you **lots** of time.
- Most programming languages have tools built for debugging code, aptly called **debuggers**.
- Python has a number of these:
  - **pdb** - the built-in debugger.
  - **DDD** - graphical front-end for many debuggers including pdb.
  - **Spyder** - Spyder has a built in debugger.
  - **PuDB** - another graphical debugger.

Debugging

The methods you use for debugging will vary depending on what kind of code you are writing, but there are some guiding principles.

Debugging

The methods you use for debugging will vary depending on what kind of code you are writing, but there are some guiding principles.

The Principle of Confirmation

"Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true about the code actually *are* true. When you find that one of your assumptions is not true, you have found a clue to the location (if not the exact nature) of a bug."
- *Fast Lane to Python*, Norm Matloff

Debugging

The methods you use for debugging will vary depending on what kind of code you are writing, but there are some guiding principles.

The Principle of Confirmation

"Fixing a buggy program is a process of confirming, one by one, that the many things you believe to be true about the code actually *are* true. When you find that one of your assumptions is not true, you have found a clue to the location (if not the exact nature) of a bug."
- *Fast Lane to Python*, Norm Matloff

In other words we are looking for things we don't expect.

Debugging Strategy

There are two main steps to debugging:

Debugging Strategy

There are two main steps to debugging:

1. **Locate** the bug.

Debugging Strategy

There are two main steps to debugging:

1. **Locate** the bug.
2. **Understand** the bug.

Debugging Strategy

There are two main steps to debugging:

1. **Locate** the bug.
2. **Understand** the bug.

Debugging Strategy

There are two main steps to debugging:

1. **Locate** the bug.
2. **Understand** the bug.

- A very common mistake made by new programmers is to try and figure out what went wrong before figuring out where it went wrong.

Debugging Strategy

There are two main steps to debugging:

1. **Locate** the bug.
2. **Understand** the bug.

- A very common mistake made by new programmers is to try and figure out what went wrong before figuring out where it went wrong.
- Localizing a bug makes understanding the bug **much** easier.

Types of Bugs: Syntax vs Semantics

There are many types of bugs. One common distinction is a syntax vs a semantic error.

- A **syntax error** is caused by typing code that the computer cannot understand.

Types of Bugs: Syntax vs Semantics

There are many types of bugs. One common distinction is a syntax vs a semantic error.

- A **syntax error** is caused by typing code that the computer cannot understand.
- For example:

Types of Bugs: Syntax vs Semantics

There are many types of bugs. One common distinction is a syntax vs a semantic error.

- A **syntax error** is caused by typing code that the computer cannot understand.
- For example:

Types of Bugs: Syntax vs Semantics

There are many types of bugs. One common distinction is a syntax vs a semantic error.

- A **syntax error** is caused by typing code that the computer cannot understand.
- For example:

```
>>> 1 !== 2
  File "<stdin>", line 1
    1 !== 2
        ^
SyntaxError: invalid syntax
```

Types of Bugs: Syntax vs Semantics

There are many types of bugs. One common distinction is a syntax vs a semantic error.

- A **syntax error** is caused by typing code that the computer cannot understand.
- For example:

```
>>> 1 !== 2
  File "<stdin>", line 1
    1 !== 2
        ^
SyntaxError: invalid syntax
```

- These will almost always cause your program to crash and are usually easy to find.

Types of Bugs: Syntax vs Semantics

- These will almost always cause your program to crash and are usually easy to find.
- A **semantic error** occurs when the code is valid, but does not do what you expect it to.
- These can be very difficult to find.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.
- Common causes include:

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.
- Common causes include:
  - Code that relies on randomness (e.g. samplers).

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.
- Common causes include:
  - Code that relies on randomness (e.g. samplers).
  - Parallel code with race conditions.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.
- Common causes include:
  - Code that relies on randomness (e.g. samplers).
  - Parallel code with race conditions.
  - Code that relies of exterior state.

Types of Bugs: Deterministic vs Non-deterministic

- A **deterministic** bug is one that will appear every time the same input is used.
- This is the most common.
- Still hard to find because they may not appear for every input.
- A **non-deterministic** bug is one that appears irregularly.
- Common causes include:
  - Code that relies on randomness (e.g. samplers).
  - Parallel code with race conditions.
  - Code that relies of exterior state.
- The most nefarious of these is the so-called **heisenbug** which disappears when you try to view it.

Debugging by Print Statements

- One very common method for accessing the intermediate state of a program is to print the values of variables or add print statements inside conditional blocks.

Debugging by Print Statements

- One very common method for accessing the intermediate state of a program is to print the values of variables or add print statements inside conditional blocks.
- We all do it, but we really shouldn't...

Debugging by Print Statements

- One very common method for accessing the intermediate state of a program is to print the values of variables or add print statements inside conditional blocks.
- We all do it, but we really shouldn't...
- As a general rule, you should not modify your code in order to debug it.

Debugging by Print Statements

- One very common method for accessing the intermediate state of a program is to print the values of variables or add print statements inside conditional blocks.
- We all do it, but we really shouldn't...
- As a general rule, you should not modify your code in order to debug it.
- In some cases modifying the code can make it so you can't reproduce the bug.

Debugging by Print Statements

- One very common method for accessing the intermediate state of a program is to print the values of variables or add print statements inside conditional blocks.
- We all do it, but we really shouldn't...
- As a general rule, you should not modify your code in order to debug it.
- In some cases modifying the code can make it so you can't reproduce the bug.
- Sometimes unavoidable.

pdb

- Instead you should use one of the many debugging tools.

pdb

- Instead you should use one of the many debugging tools.
- The built-in Python debugger is called `pdb`.

pdb

- Instead you should use one of the many debugging tools.
- The built-in Python debugger is called `pdb`.

pdb

- Instead you should use one of the many debugging tools.
- The built-in Python debugger is called `pdb`.

```
$> python -m pdb script.py
> /Users/adams/Dropbox/590n/demos/script.py(1)<module>()
-> import numpy as np
(Pdb)
```

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

```
(Pdb) b 8
Breakpoint 1 at /Users/adams/Dropbox/590n/demos/script.
    py:8
(Pdb)
```

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

```
(Pdb) b 8
Breakpoint 1 at /Users/adams/Dropbox/590n/demos/script.
    py:8
(Pdb)
```

- To run the script, use the `c` or `continue` command.

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

```
(Pdb) b 8
Breakpoint 1 at /Users/adams/Dropbox/590n/demos/script.
    py:8
(Pdb)
```

- To run the script, use the `c` or `continue` command.

Breakpoints

- **Breakpoints** allow you to stop the code at a specified line and interactively examine the state.
- In pdb, breakpoints are set with the command `b` or `break` followed by a line number.

```
(Pdb) b 8
Breakpoint 1 at /Users/adams/Dropbox/590n/demos/script.
    py:8
(Pdb)
```

- To run the script, use the `c` or `continue` command.

```
(Pdb) c
> /Users/adams/Dropbox/590n/demos/script.py(8)<module>()
-> b = 3
(Pdb)
```

Viewing Variables

- Once you are inside the code, you can explore the state (that is, the current values of the variables).

Viewing Variables

- Once you are inside the code, you can explore the state (that is, the current values of the variables).
- Print variables using the `p` or `print` command.

Viewing Variables

- Once you are inside the code, you can explore the state (that is, the current values of the variables).
- Print variables using the `p` or `print` command.

Viewing Variables

- Once you are inside the code, you can explore the state (that is, the current values of the variables).
- Print variables using the `p` or `print` command.

```
(Pdb) c
> /Users/adams/Dropbox/590n/demos/script.py(8)<module>()
-> b = 3
(Pdb) p a
2
```

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.
- For example, if we want to stop inside a loop but only on the last iteration.

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.
- For example, if we want to stop inside a loop but only on the last iteration.
- A **conditional breakpoint** will halt the code only when a logical condition returns True.

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.
- For example, if we want to stop inside a loop but only on the last iteration.
- A **conditional breakpoint** will halt the code only when a logical condition returns True.
- This condition may involve variables defined in the code.

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.
- For example, if we want to stop inside a loop but only on the last iteration.
- A **conditional breakpoint** will halt the code only when a logical condition returns True.
- This condition may involve variables defined in the code.

Conditional Breakpoints

- It is often convenient to view the state only when a certain condition is met.
- For example, if we want to stop inside a loop but only on the last iteration.
- A **conditional breakpoint** will halt the code only when a logical condition returns True.
- This condition may involve variables defined in the code.

```
# Break at line <line_number> when <condition> is True
(Pdb) b <line_number>, <condition>
```

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.
- The `n` or `next` command will advance the code to the next line **in the current scope**. It will not enter functions calls.

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.
- The `n` or `next` command will advance the code to the next line **in the current scope**. It will not enter functions calls.
- The `s` or `step` command will advance the code to the next line and will enter function calls.

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.
- The `n` or `next` command will advance the code to the next line **in the current scope**. It will not enter functions calls.
- The `s` or `step` command will advance the code to the next line and will enter function calls.
- The `until` command will run the code until the specified line.

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.
- The `n` or `next` command will advance the code to the next line **in the current scope**. It will not enter functions calls.
- The `s` or `step` command will advance the code to the next line and will enter function calls.
- The `until` command will run the code until the specified line.

Stepping Through Code

- Once you have reached a breakpoint, it is often useful to step through the code one line at a time.
- The `n` or `next` command will advance the code to the next line **in the current scope**. It will not enter functions calls.
- The `s` or `step` command will advance the code to the next line and will enter function calls.
- The `until` command will run the code until the specified line.

```
(Pdb) s
(Pdb) n
(Pdb) until <line_number>
```

Outline

Exceptions

- When Python encounters a run time error, it raises an **exception**.

Exceptions

- When Python encounters a run time error, it raises an **exception**.
- By default this will crash your code.

Exceptions

- When Python encounters a run time error, it raises an **exception**.
- By default this will crash your code.
- There are many cases where we might want to keep running:

Exceptions

- When Python encounters a run time error, it raises an **exception**.
- By default this will crash your code.
- There are many cases where we might want to keep running:
  - Continuous code (e.g. a server application).

Exceptions

- When Python encounters a run time error, it raises an **exception**.
- By default this will crash your code.
- There are many cases where we might want to keep running:
  - Continuous code (e.g. a server application).
  - Restart algorithm with a new initialization.

Exceptions

- When Python encounters a run time error, it raises an **exception**.
- By default this will crash your code.
- There are many cases where we might want to keep running:
  - Continuous code (e.g. a server application).
  - Restart algorithm with a new initialization.
  - Solicit the user for input.

Exceptions

- The Python **try-except** block allows you to gracefully recover from an error.

Exceptions

- The Python **try-except** block allows you to gracefully recover from an error.

Exceptions

- The Python **try-except** block allows you to gracefully recover from an error.

```
try:
    <code_that_might_have_errors>
except:
    <error_handling_code>
```

Exceptions

- The Python **try-except** block allows you to gracefully recover from an error.

```
try:
    <code_that_might_have_errors>
except:
    <error_handling_code>
```

- Python will try to run the code in the try block and if it fails, it will run the code in the except block.

Exceptions

- Sometimes errors contain information and you may want to handle different exceptions in different ways.

Exceptions

- Sometimes errors contain information and you may want to handle different exceptions in different ways.

Exceptions

- Sometimes errors contain information and you may want to handle different exceptions in different ways.

```
try:
    <code_that_might_have_errors>
except <exception_type1>:
    <type1_handler>
except <exception_type2>:
    <type2_handler>
except:
    <handle_everything_else>
```

Exceptions

- Common built-in exception types include:

Exceptions

- Common built-in exception types include:
  - `ValueError`

Exceptions

- Common built-in exception types include:
  - `ValueError`
  - `ArithmeticError`

Exceptions

- Common built-in exception types include:
  - `ValueError`
  - `ArithmeticError`
  - `IOError`

Exceptions

- Common built-in exception types include:
  - `ValueError`
  - `ArithmeticError`
  - `IOError`
  - `KeyError`

Exceptions

- Common built-in exception types include:
  - `ValueError`
  - `ArithmeticError`
  - `IOError`
  - `KeyError`
  - etc.

Exceptions

You can handle different errors in different ways:

```
try:
    <code_that_might_have_errors>
except <exception_type1>:
    <type1_handler>
except <exception_type2>:
    <type2_handler>
except:
    <handle_everything_else>
```

Raising Exceptions

- Python also allows you to **raise** your own exceptions.

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).
  - Allows the person using your code to handle the exception gracefully.

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).
  - Allows the person using your code to handle the exception gracefully.
  - You will probably forget all of the ways your code can fail...

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).
  - Allows the person using your code to handle the exception gracefully.
  - You will probably forget all of the ways your code can fail...
- Raise exceptions using the `raise` command.

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).
  - Allows the person using your code to handle the exception gracefully.
  - You will probably forget all of the ways your code can fail...
- Raise exceptions using the `raise` command.

Raising Exceptions

- Python also allows you to **raise** your own exceptions.
- Reasons you may want to do this include:
  - It provides information about the error (as opposed to failing silently).
  - Allows the person using your code to handle the exception gracefully.
  - You will probably forget all of the ways your code can fail...
- Raise exceptions using the `raise` command.

```
# Raise a value error
raise ValueError("Cannot invert a singular matrix.")
```

Cleaning Up

- Sometimes there is code you want to run regardless of whether there was an error or not.

Cleaning Up

- Sometimes there is code you want to run regardless of whether there was an error or not.
- This code can be written in a **finally** clause.

Cleaning Up

- Sometimes there is code you want to run regardless of whether there was an error or not.
- This code can be written in a **finally** clause.
- Code in a finally block with run whether or not the code in the try block succeeds.

Cleaning Up

- Sometimes there is code you want to run regardless of whether there was an error or not.
- This code can be written in a **finally** clause.
- Code in a finally block with run whether or not the code in the try block succeeds.

Cleaning Up

- Sometimes there is code you want to run regardless of whether there was an error or not.
- This code can be written in a **finally** clause.
- Code in a finally block with run whether or not the code in the try block succeeds.

```
try:
    <code_that_might_have_errors>
except:
    <error_handling_code>
finally:
    <always_run_this>
```

Exception Handling is Not Just For Exceptions

Try-except blocks can be used for many things besides handling errors. For example:

Exception Handling is Not Just For Exceptions

Try-except blocks can be used for many things besides handling errors. For example:

```
a = {}
try:
    a["key"].append(5)
except:
    a["key"] = [5]
```

Exception Handling is Not Just For Exceptions

Try-except blocks can be used for many things besides handling errors. For example:

```
a = {}
try:
    a["key"].append(5)
except:
    a["key"] = [5]
```

- Here we are try to create a dictionary of lists.
- We can use try-except to check if a list is in the dictionary and add it if it is not.