Programming Principles in Python (CSCI 503/490)

Debugging & Testing

Dr. David Koop

Multiple Inheritance

- Can have a class inherit from two different superclasses
- HybridCar inherits from Car and Hybrid
- Python allows this!
  - `class HybridCar(Car, Hybrid): …`
- Problem: how is `super()` is defined?
  - Diamond Problem
  - Python use the **method resolution order** (MRO) to determine order of calls
- Method resolution order:
  - `mro()` is a **class** method and **order** of superclasses matters
  - `Square.mro() # [__main__.Square, __main__.Rectangle, object]`

Duck Typing

- "If it looks like a duck and quacks like a duck, it must be a duck."
- Python "does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used"
-
```
class Rectangle:
    def area(self):
        …
```
-
```
class Circle:
    def area(self):
        …
```
- It doesn't matter that they don't have a common base class as long as they respond to the methods/attributes we expect: `shape.area()`

[Python Glossary]

Type Annotations

- ```
  def area(width : float, height : float) -> float:
      return width * height
  ```
- Type annotations **do not** affect runtime behavior!
  - `area("abc", 3) # runs, returns "abcabcabc"`
- Can also specify types for variables including nested data structures
  - ```
    from typing import List
    names : List[str] = ['Alice', 'Bob']
    ```
- Tools like `mypy` can be used to do static type checking
- Pros & Cons:
  - Good for libraries, help to specify APIs and improve
  - Take additional time, not generally used for scripts

Dataclasses

- `namedtuple` and `SimpleNamespace` store data with dot access (.field)
- Dataclasses simply boilerplate tasks (constructor, repr, comparison methods)
- Specify type annotations on class attributes, decorator creates class
- Example:

```
- from dataclasses import dataclass
  @dataclass
  class Rectangle:
      width: float
      height: float
- Rectangle(34, 21) # just works!
```

Advantages of Exceptions

- Separate error-handling code from "regular" code
  - Too many potential errors to check
  - Don't have to check even possible error in the code
- Programmer decides when to handle the exceptions
  - Allows propagation of errors up the call stack
  - Errors can be grouped and differentiated

[Java Tutorial, Oracle]

Try-Except

- The `try` statement has the following form:

```
try:
   <body>
except <ErrorType>*:
   <handler>
```

- When Python encounters a `try` statement, it attempts to execute the statements inside the body.
- If there is no error, control passes to the next statement after the `try`... `except` (unless `else` or `finally` clauses)
- Note: **except** not catch

Exception Granularity

- If you catch any exception using a base class near the top of the hierarchy, you may be **masking** code errors
- ```
  try:
      c, d = a / b
  except Exception:
      c, d = 0, 0
  ```
- Remember `Exception` catches any exception is an instance of Exception
- Catches `TypeError: cannot unpack non-iterable float object`
- Better to have more **granular** (specific) exceptions!
- We don't want to catch the TypeError because this is a **programming error** not a runtime error
  ```

Assignment 5

- Due Today
- Similar Pokémon entry data as A3, but **different** dataset
- The data should be included in the python package
- Scripts, modules, packages
- Command-line program

Assignment 6

- Object-Oriented Programming
- Classes to create an online store
  - Operators
  - Representations
  - Exceptions

Test 2

- Wednesday, November 5, in class from 9:30-10:45am
- Similar Format to Test 1
- Emphasizes topics covered since Test 1, but still need to know core concepts from the first third of the course

Exception Locality

- Generally, want try statement to be specific to a part of the code
-
```
try:
    with open('missing-file.dat') as f:
        lines = f.readlines()
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print("An error occurred processing files.")
```
- We don't know whether reading failed or writing failed
- Maybe that is ok, but having multiple try-except clauses might help

[Deitel & Deitel]

Exception Locality

```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
except OSError:
    print(f"An error occurred reading {fname}")
try:
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print(f"An error occurred writing {out_fname}")
```

Multiple Except Clauses

- May also be able to address with **multiple** except clauses:
- ```
  try:
      fname = 'missing-file.dat'
      with open(fname) as f:
          lines = f.readlines()
      out_fname = 'output-file.dat'
      with open('output-file.dat', 'w') as fout:
          fout.write("Testing")
  except FileNotFoundError:
      print(f"File {fname} does not exist")
  except PermissionError:
      print(f"Cannot write to {out_fname}")
  ```
- However, other OSError problems (disk full, etc.) won't be caught
  ```

Multiple Except Clauses

- Function like an if/elif sequence
- Checked in order so put more granular exceptions earlier!
-
```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except OSError:
    print("An error occurred processing files")
```

Multiple Except Clauses

- Function like an if/elif sequence
- Checked in order so put more granular exceptions **earlier**!
-
```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError:
    print("An error occurred processing files")
except FileNotFoundError:
    print(f"File {fname} does not exist")
```

Multiple Except Clauses

- Function like an if/elif sequence
- Checked in order so put more granular exceptions **earlier**!
- ```
  try:
      fname = 'missing-file.dat'
      with open(fname) as f:
          lines = f.readlines()
      out_fname = 'output-file.dat'
      with open('output-file.dat', 'w') as fout:
          fout.write("Testing")
  except OSError:
      print("An error occurred processing files")
  except FileNotFoundError:
      print(f"File {fname} does not exist")
  ```
  ```

Bare Except

- The bare except clause acts as a catch-all (elif any other exception)
-
```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except FileNotFoundError:
    print(f"File {fname} does not exist")
except OSError:
    print("An error occurred processing files")
except:
    print("Any other error goes here")
```

Handling Multiple Exceptions at Once

- Can process multiple exceptions with one clause, use **tuple** of classes
- Allows some specificity but without repeating
- ```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except (FileNotFoundError, PermissionError):
    print("An error occurred processing files")
```
```

Exception Objects

- Exceptions themselves are a type of object.
- If you follow the error type with an identifier in an except clause, Python will assign that identifier the actual exception object.
- Sometimes exceptions encode information that is useful for handling
- ```
try:
    fname = 'missing-file.dat'
    with open(fname) as f:
        lines = f.readlines()
    out_fname = 'output-file.dat'
    with open('output-file.dat', 'w') as fout:
        fout.write("Testing")
except OSError as e:
    print(e.errno, e.filename, e)
```
```

Else Clause

- Code that executes if no exception occurs
- ```
  b = 3
  a = 2
  try:
      c = b / a
  except ZeroDivisionError:
      print("Division failed")
      c = 0
  else:
      print("Division successful:", c)
  ```
  ```

Finally

- Code that always runs, **regardless** of whether there is an exception
- ```
  b = 3
  a = 0
  try:
      c = b / a
  except ZeroDivisionError:
      print("Division failed")
      c = 0
  finally:
      print("This always runs")
  ```
  ```

Finally

- Code that always runs, **regardless** of whether there is an exception
- …even if the exception isn't handled!
- ```
  b = 3
  a = 0
  try:
      c = b / a
  finally:
      print("This always runs, even if we crash")
  ```
- Remember that context managers (e.g. for files) have built-in cleanup clauses
  ```

Nesting

- You can nest try-except clauses inside of except clauses, too.
- Example: perhaps a file load could fail so you want to try an alternative location but want to know if that fails, too.
- Can even do this in a `finally` clause:
- ```
  try:
      c = b / a
  finally:
      try:
          print("This always runs", 3/0)
      except ZeroDivisionError:
          print("It is silly to only catch this exception")
  ```
  ```

Raising Exceptions

- Create an exception and raise it using the `raise` keyword
- Pass a string that provides some detail
- Example: `raise Exception("This did not work correctly")`
- Try to find a exception class:
  - `ValueError`: if an argument doesn't fit the function's expectations
  - `NotImplementedError`: if a method isn't implemented (e.g. abstract cls)
- Be specific in the error message, state actual values
- Can also subclass from existing exception class, but check if existing exception works first
- Some packages create their own base exception class (`RequestException`)

Re-raising and Raising From

- Sometimes, we want to detect an exception but also pass it along
- ```
  try:
      c = b / a
  except ZeroDivisionError:
      print("Division failed")
      raise
  ```
- Raising from allows exception to show specific chain of issues
- ```
  try:
      c = b / a
  except ZeroDivisionError as e:
      print("Division failed")
      raise ValueError("a cannot be zero") from e
  ```
- Usually unnecessary because Python does the right thing here (shows chain)

Making Sense of Exceptions

- When code (e.g. a cell) crashes, read the traceback:
-
```
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-58-488e97ad7d74> in <module>
      4     return divide(a+b, a-b)
      5 for i in range(4):
----> 6     process(3, i)
<ipython-input-58-488e97ad7d74> in process(a, b)
      3         return c / d
----> 4     return divide(a+b, a-b)
      5 for i in range(4):
<ipython-input-58-488e97ad7d74> in divide(c, d)
      2     def divide(c, d):
----> 3         return c / d
      4     return divide(a+b, a-b)
ZeroDivisionError: division by zero
```

Making Sense of Exceptions

- Start at the bottom: last line is the exception message
- Nesting goes outside-in: innermost scope is last, outermost scope is first
- Arrows point to the line of code that caused errors at each scope
- Surrounding lines give context

Making Sense of Exceptions

- Sometimes, exception handling can mask actual issue!
- ```
  def process(a, b):
      …
  for i in range(4):
      try:
          process(3, i)
      except ZeroDivisionError:
          raise Exception(f"Cannot process i={i}") from None
  ```
- ```
  Exception                          Traceback (most recent call last)
  <ipython-input-60-6d0289010945> in <module>
        7         process(3, i)
        8     except ZeroDivisionError:
  ----> 9         raise Exception(f"Cannot process i={i}") from None
  Exception: Cannot process i=3
  ```
- Usually, Python includes inner exception (`from None` stops the chain)

Making Sense of Exceptions

- Probably the **worst** thing is to **ignore** all exceptions:
- ```
  def process(a, b):
      …
  result = []
  for i in range(6):
      try:
          result.append(process(3, i))
      except:
          pass
  ```
- This may seem like the easy way out, don't have to worry about errors, but can mask major issues in the code!
- Be specific (granularity), try to handle cases when something goes wrong, crash **gracefully** if it is an unexpected error
  ```

Python 3.11: Fine-Grained Error Locations

- Code is faster (10-60% faster than 3.10, 25% average on benchmark)
- Debugging: Errors can show more specific locations
- Old Error:

```
- Traceback (most recent call last):
    File "distance.py", line 11, in <module>
      print(manhattan_distance(p1, p2))
    File "distance.py", line 6, in manhattan_distance
      return abs(pt_1.x - pt_2.x) + abs(pt_1.y - pt_2.y)
  AttributeError: 'NoneType' object has no attribute 'x'
```

Python 3.11: Fine-Grained Error Locations

- New Error:

```
- Traceback (most recent call last):
    File "distance.py", line 11, in <module>
      print(manhattan_distance(p1, p2))
            ^^^^^^^^^^^^^^^^^^^^^^^^^^
    File "distance.py", line 6, in manhattan_distance
      return abs(pt_1.x - pt_2.x) + abs(pt_1.y - pt_2.y)
                         ^^^^^^
  AttributeError: 'NoneType' object has no attribute 'x'
```

Python 3.11: Fine-Grained Error Locations

```
Traceback (most recent call last):
  File "query.py", line 37, in <module>
    magic_arithmetic('foo')
  File "query.py", line 18, in magic_arithmetic
    return add_counts(x) / 25
           ^^^^^^^^^^^^^^
  File "query.py", line 24, in add_counts
    return 25 + query_user(user1) + query_user(user2)
                ^^^^^^^^^^^^^^^^^
  File "query.py", line 32, in query_user
    return count(db, response['a']['b']['c']['user'])
                     ~~~~~~~~~~~~~~~~~~~~^^^^^
TypeError: 'NoneType' object is not subscriptable
```

How do you debug code?

Debugging

- print statements
- logging library
- pdb
- Extensions for IDEs (e.g. PyCharm)
- JupyterLab Debugger Support

Print Statements

- Just print the values or other information about identifiers:
- ```
  def my_function(a, b):
      print(a, b)
      print(b - a == 0)
      return a + b
  ```
- Note that we need to remember what is being printed
- Can add this to print call, or use f-strings with trailing = which causes the name and value of the variable to be printed
- ```
  def my_function(a, b):
      print(f"{a=} {b=} {b - a == 0}")
      return a + b
  ```

Print Problems

- Have to uncomment/comment
- Have to remember to get rid of (or comment out) debugging statements when publishing code
- Print can dump a lot of text (slows down notebooks)
- Can try to be smarter:

```
- if i % 100 == 0:
      print(i, f"{current_output=}")
- do_print = value == 42
  if do_print:
      print(f"{a=} {current_output=}")
```

Logging Library

- Allows different levels of output (e.g. DEBUG, INFO, WARNING, ERROR CRITICAL)
- Can output to a file as well as stdout/stderr
- Can configure to suppress certain levels or filter messages
- ```
  import logging
  def my_function(a,b):
      logging.debug(f"{a=} {b=} {b-a == 0}")
      return a + b
  my_function(3, 5)
  ```
- This doesn't work in notebooks…
  ```

Logging Library

- Need to set default level (e.g. DEBUG)
- For notebooks, best to define own logger and set level
- ```
  import logging
  logger = logging.Logger('my-logger')
  logger.setLevel(logging.DEBUG)
  def my_function(a,b):
      logger.debug(f"{a=} {b=} {b-a == 0}")
      return a + b
  my_function(3, 5)
  ```
- Prints on stderr, can set to stdout via:
- ```
  import sys
  logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)
  ```

Python Debugger (pdb)

- Debuggers offer the ability to inspect and interact with code as it is running
  - Define breakpoints as places to stop code and enter the debugger
  - Commands to inspect variables and step through code
  - Different types of steps (into, over, continue)
  - Can have multiple breakpoints in a piece of code
- There are a number of debuggers like those built into IDEs (e.g. PyCharm)
- pdb is standard Python, also an ipdb variant for IPython/notebooks

Python Debugger

- Post-mortem inspection:
  - In the notebook, use `%debug` in a new cell to inspect at the line that raised the exception
    - Can have this happen all the time using `%pdb` magic
    - Brings up a new panel that allows debugging interactions
  - In a script, run the script using pdb:
    - `python -m pdb my_script.py`

Python Debugger

- Breakpoints
  - To set a breakpoint, simply add a `breakpoint()` call in the code
  - Before Python 3.7, this required `import pdb; pdb.set_trace()`
  - Run the cell/script as normal and pdb will start when it hits the breakpoint

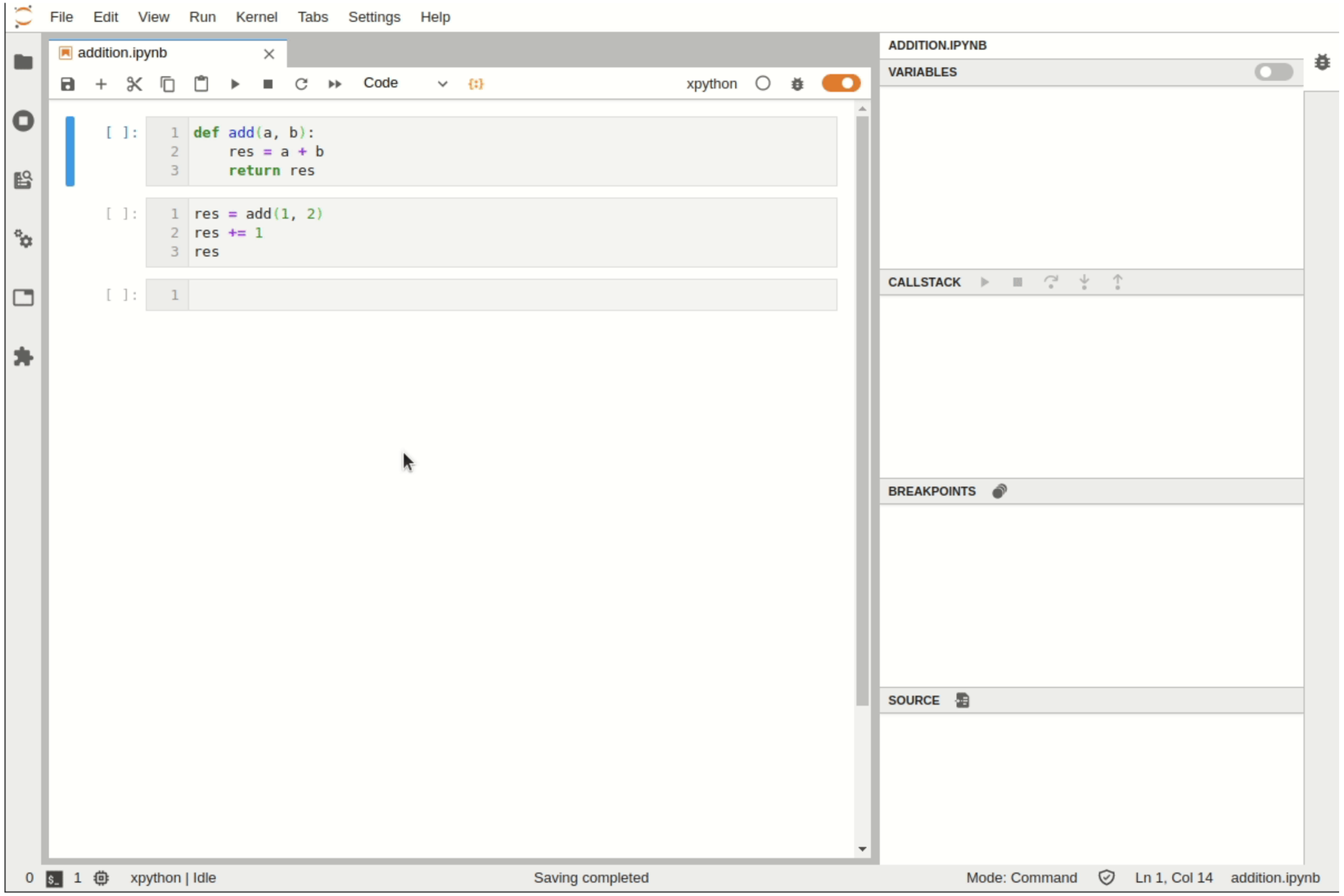
```
> <ipython-input-1-792bb5fe2598>(3)divide()
      1 def process(a, b):
      2     def divide(c, d):
----> 3         return c / d
      4     return divide(a+b, a-b)
      5 result = []

ipdb>
```

Python Debugger Commands

- `p` [print expressions]: Print expressions, comma separated
- `n` [step over]: continue until next line in **current function**
- `s` [step into]: stop at next line of code (same function or one being called)
- `c` [continue]: continue execution until next breakpoint
- `l` [list code]: list source code (ipdb does this already), also `ll` (fewer lines)
- `b` [breakpoints]: list or set new breakpoint (with line number)
- `w` [print stack trace]: Prints the stack (like what notebook shows during traceback), `u` and `d` commands move up/down the stack
- `q` [quit]: quit
- `h` [help]: help (there are many other commands)

Jupyter Debugging Support



Jupyter Debugging Support