CSC148 winter 2014
inheritance, Exceptions, special methods
week 3

Danny Heap (with additions by Dustin Wehr)
heap@cs.toronto.edu (dustin.wehr@utoronto.ca)
BA4270, behind elevators (SF4306D)
http://www.cdf.toronto.edu/~heap/148/W14/

January 23, 2014
Topics today

- Specializing software
  - inheritance
  - extending vs. overriding
  - calling superclass constructors (special case of \_\_init\_\_)
- Exceptions
  - what they are
  - why we use them
  - raising
  - catching ("except" clause)
  - defining your own
from previous weeks

Confused/worried about properties?
https://piazza.com/class/hqaccaidcrq44o?cid=88

Very uncomfortable with recursion?
https://piazza.com/class/hqaccaidcrq44o?cid=94
Why have a Queue class...

...when list objects can do everything Queue objects can do, plus more?
Why have a Queue class...

...when list objects can do everything Queue objects can do, plus more?

- Hard-to-ignore communication of the programmer’s intentions
documentation, basically
Why have a Queue class...

...when list objects can do everything Queue objects can do, plus more?

- Hard-to-ignore communication of the programmer’s intentions documentation, basically
- More-efficient implementation
If we decided to extend the features of Stack, what’s wrong with:

- modifying the existing Stack?

- copy-paste-modify Stack \(\rightarrow\) MyStack?

- include Stack attribute in new classes
specialize flexibly

If we decided to extend the features of Stack, what’s wrong with:

- modifying the existing Stack?
  – Ok if there was something you should have put there in the first place. But what if we want to extend the features of Stack in two different, incompatible ways?

- copy-paste-modify Stack \(\rightarrow\) MyStack?

- include Stack attribute in new classes
specialize flexibly

If we decided to extend the features of Stack, what’s wrong with:

- modifying the existing Stack?
  - Ok if there was something you should have put there in the first place. But what if we want to extend the features of Stack in two different, incompatible ways?
  - *Code/feature bloat:* introduces unnecessary complication/clutter for users for whom the original Stack class was adequate.

- copy-paste-modify Stack → MyStack?

- include Stack attribute in new classes
specialize flexibly

If we decided to extend the features of Stack, what’s wrong with:

- modifying the existing Stack?
  - Ok if there was something you should have put there in the first place. But what if we want to extend the features of Stack in two different, incompatible ways?
  - *Code/feature bloat*: introduces unnecessary complication/clutter for users for whom the original Stack class was adequate.

- copy-paste-modify Stack $\rightarrow$ MyStack?
  - Improvements/fixes of Stack will need to be repeated in MyStack.

- include Stack attribute in new classes
specialize flexibly

If we decided to extend the features of Stack, what’s wrong with:

➤ modifying the existing Stack?
   — Ok if there was something you should have put there in the first place. But what if we want to extend the features of Stack in two different, incompatible ways?
   — Code/feature bloat: introduces unnecessary complication/clutter for users for whom the original Stack class was adequate.

➤ copy-paste-modify Stack → MyStack?
   Improvements/fixes of Stack will need to be repeated in MyStack.

➤ include Stack attribute in new classes
   Will work in some cases, but limited since we can’t change anything about the internal representation of the stack.
class declaration

we subclass (extend) a superclass (base class) by:

- declaring that we’re extending it...
  ```python
  class NewClass(OldClass):
    ...
  ```

- add methods and attributes to specialize

- other methods and attributes are searched for in superclass
override versus extend

you may replace or modify old code

- subclass method with the same name replaces superclass method

- access superclass method with `OldClass.method(self,...)`

- `__init__` is a special case — careful
exceptions: richer communication

return types are not appropriate in all cases

➤ what’s wrong with IntStack returning a “special” integer for pop-on-empty? Or returning None?

➤ push usually has return type None, but what if stuff happens?

➤ what if the calling code doesn’t know what to do?
cause existing Exceptions:

- `int("seven")`

- `a = 1/0`

- `[1, 2][2]`
cause existing Exceptions:

- `int("seven")`
  
  `builtins.ValueError: invalid literal for int() with base 10: 'seven'`

- `a = 1/0`

- `[1, 2][2]`
cause existing Exceptions:

- `int("seven")`
  
builtins.ValueError: invalid literal for int() with base 10: 'seven'

- `a = 1/0`
  
builtins.ZeroDivisionError: division by zero

- `[1, 2][2]`
cause existing Exceptions:

- `int("seven")`
  builtins.ValueError: invalid literal for int() with base 10: ‘seven’

- `a = 1/0`
  builtins.ZeroDivisionError: division by zero

- `[1, 2][2]`
  builtins.IndexError: list index out of range
raise existing Exceptions:

- raise ValueError or...

- raise ValueError("you can’t do that!")
roll your own Exceptions:

- class ExtremeException(Exception):
  pass

- raise ExtremeException

- raise ExtremeException('I really take exception to that!')
exceptions: separation of concerns

- Suppose we’re writing a chat client.
- We’re fine with telling users that a prerequisite for using the client *at all* is that you’re connected to the internet.
- *Many* places in the code where we need to do network communication, which will fail if user is not connected to the internet.
- We can define a new type of exception (or use a built-in one) that gets raised in many places but handled in one place.

```python
# ConnectionError is a built-in subclass of Exception
if __name__ == '__main__':
    running = True
    while running:
        try:
            con = establish_connection()
            run_with_connection(con)
        except ConnectionError
            system.wait(5)  # wait 5 seconds before trying again
            # todo: notify user, and increase parameter 5 each time
```
what makes two stack equivalent?

Tell Python with `__eq__`

Your `__eq__` should really be equivalent: symmetrical, reflexive, transitive

- Transitivity is the easiest property to accidentally get wrong.
represent in a reproducible way

Tell Python how to represent your object with \texttt{\_\_repr\_\_}

Ideally, you should be able to cut-and-paste this representation to create an equivalent object
extras 1: Nameless functions with lambda

—we didn’t look at this slide in class, but we’ll be covering this later in the semester—

Writing \((\lambda x: \text{one-line-function-body})\) in a given place in your code accomplishes the same thing as first defining a function

```python
def fn_name(x):
    \text{one-line-function-body}
```

and then writing `fn_name` in that same place in your code.

```python
def square(x:int):
    return x**2
```

```python
print(square(5))
print((\lambda x: x**2)(5))
```

Nothing deep!

It is simply more-concise and doesn’t require you to introduce a name for the function, which is good *if you’re only going to use the function once.*
extras 2: Useful built-in functions to use with lambda

—we didn’t look at this slide in class, but we’ll be covering this later in the semester—

- **filter(f, iterable_object)** returns an object of the same type as iterable_object that contains only the elements \( x \in \text{iterable\_object} \) such that \( f(x) \) return true.

  What’s this do?

  ```python
  filter(lambda x: x > 0, [1, 0, 4, -1])
  ```

- **map(f, iterable_object)** returns an object of the same type and size as iterable_object obtained by applying the function \( f \) to each of iterable_object.

  What’s this do?

  ```python
  map(lambda x: x**2, [1, 0, 4, -1])
  ```

  You already know this one! Same as

  ```python
  [x**2 for x in [1,0,4,-1]]
  ```