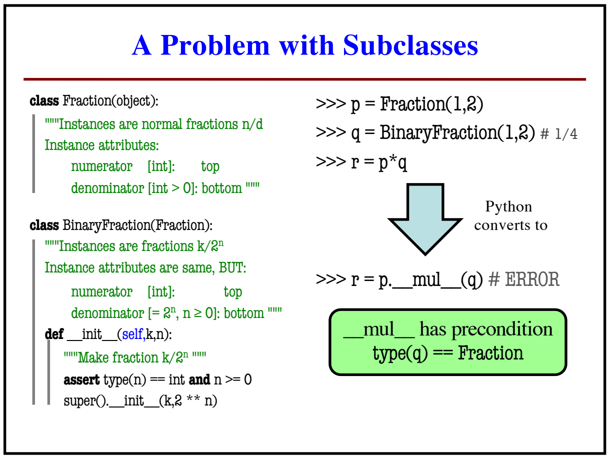


A Problem with Subclasses

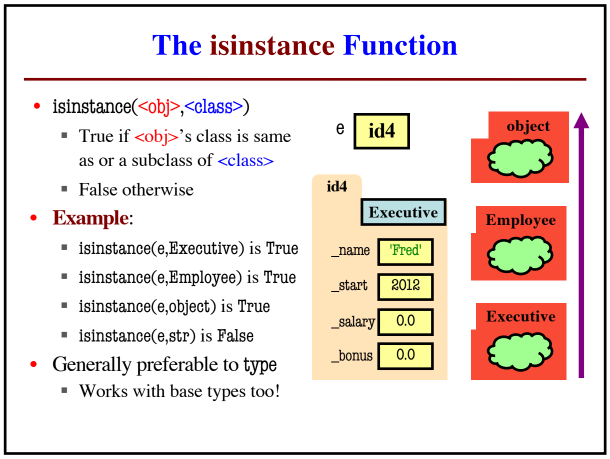
```
class Fraction(object):
    """Instances are normal fractions n/d
    Instance attributes:
        numerator    [int]:        top
        denominator [int > 0]: bottom """

class BinaryFraction(Fraction):
    """Instances are fractions k/2^n
    Instance attributes are same, BUT:
        numerator    [int]:              top
        denominator [= 2^n, n ≥ 0]: bottom """
    def __init__(self,k,n):
        """Make fraction k/2^n """
        assert type(n) == int and n >= 0
        super().__init__(k,2 ** n)
```

```
>>> p = Fraction(1,2)
>>> q = BinaryFraction(1,2) # 1/4
>>> r = p*q
```

Python converts to

```
>>> r = p.__mul__(q) # ERROR
```

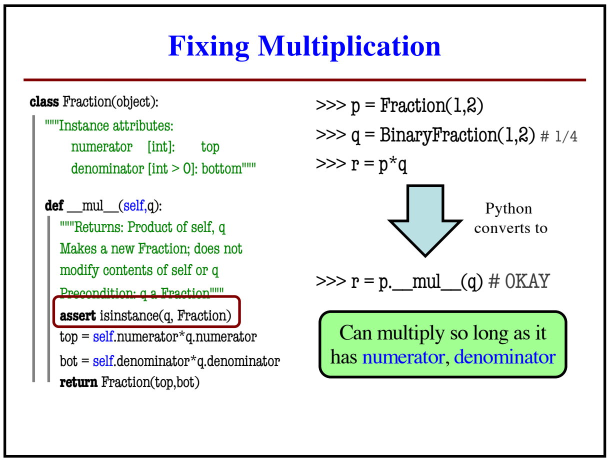
__mul__ has precondition
type(q) == Fraction



The isinstance Function

- isinstance(<obj>,<class>)
  - True if <obj>'s class is same as or a subclass of <class>
  - False otherwise
- **Example**:
  - isinstance(e,Executive) is True
  - isinstance(e,Employee) is True
  - isinstance(e,object) is True
  - isinstance(e,str) is False
- Generally preferable to type
  - Works with base types too!

e
id4
id4
Executive
_name 'Fred'
_start 2012
_salary 0.0
_bonus 0.0
object
Employee
Executive



Fixing Multiplication

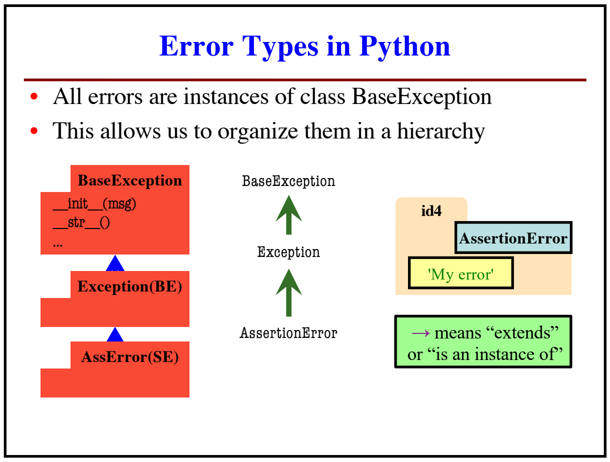
```
class Fraction(object):
    """Instance attributes:
        numerator    [int]:        top
        denominator [int > 0]: bottom"""

    def __mul__(self,q):
        """Returns: Product of self, q
        Makes a new Fraction; does not
        modify contents of self or q
        ~~Precondition: q a Fraction~~"""
        assert isinstance(q, Fraction)
        top = self.numerator*q.numerator
        bot = self.denominator*q.denominator
        return Fraction(top,bot)
```

```
>>> p = Fraction(1,2)
>>> q = BinaryFraction(1,2) # 1/4
>>> r = p*q
```

Python converts to

```
>>> r = p.__mul__(q) # OKAY
```

Can multiply so long as it has numerator, denominator

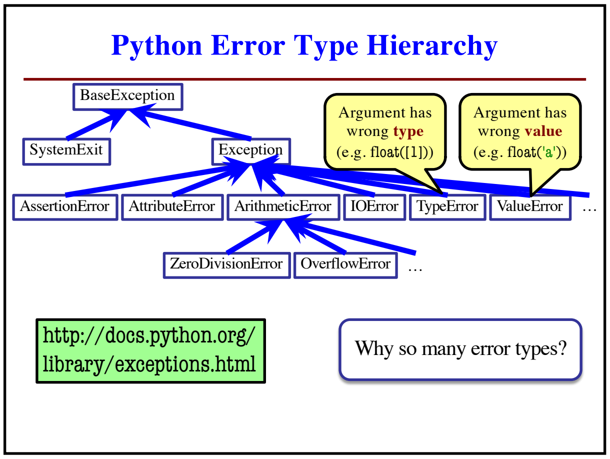


Error Types in Python

- All errors are instances of class BaseException
- This allows us to organize them in a hierarchy

BaseException
__init__(msg)
__str__()
...
Exception(BE)
AssError(SE)
BaseException
Exception
AssertionError
id4
AssertionError
'My error'

→ means "extends" or "is an instance of"



Python Error Type Hierarchy

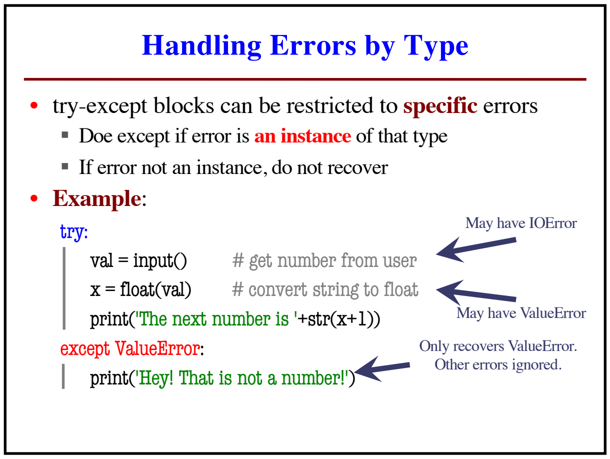
BaseException
SystemExit
Exception
AssertionError
AttributeError
ArithmeticError
IOError
TypeError
ValueError
…
ZeroDivisionError
OverflowError
…

Argument has wrong **type** (e.g. float([1]))

Argument has wrong **value** (e.g. float('a'))

http://docs.python.org/library/exceptions.html

Why so many error types?



Handling Errors by Type

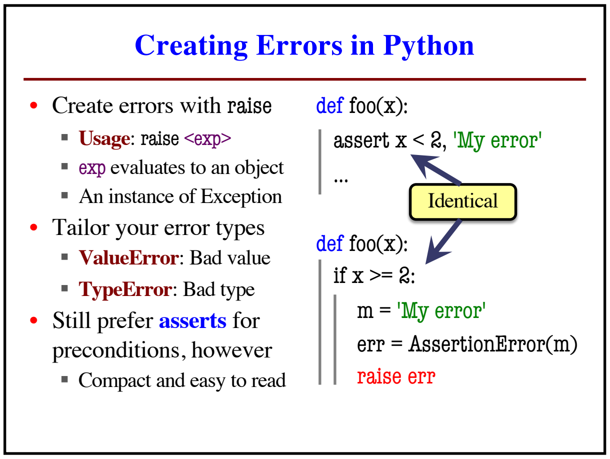
- try-except blocks can be restricted to **specific** errors
  - Doe except if error is **an instance** of that type
  - If error not an instance, do not recover
- **Example**:

```
try:
    val = input()        # get number from user
    x = float(val)       # convert string to float
    print('The next number is '+str(x+1))
except ValueError:
    print('Hey! That is not a number!')
```

May have IOError

May have ValueError

Only recovers ValueError. Other errors ignored.



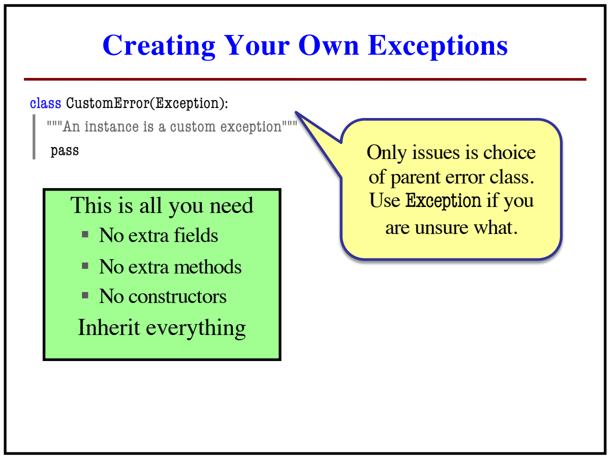
Creating Errors in Python

- Create errors with raise
  - **Usage**: raise <exp>
  - exp evaluates to an object
  - An instance of Exception
- Tailor your error types
  - **ValueError**: Bad value
  - **TypeError**: Bad type
- Still prefer **asserts** for preconditions, however
  - Compact and easy to read

```
def foo(x):
    assert x < 2, 'My error'
    ...
```

Identical

```
def foo(x):
    if x >= 2:
        m = 'My error'
        err = AssertionError(m)
        raise err
```



Creating Your Own Exceptions

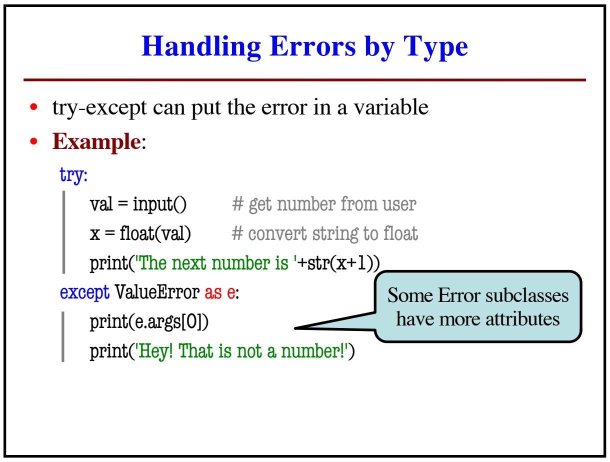
```
class CustomError(Exception):
    """An instance is a custom exception"""
    pass
```

Only issues is choice of parent error class. Use Exception if you are unsure what.

This is all you need

- No extra fields
- No extra methods
- No constructors

Inherit everything

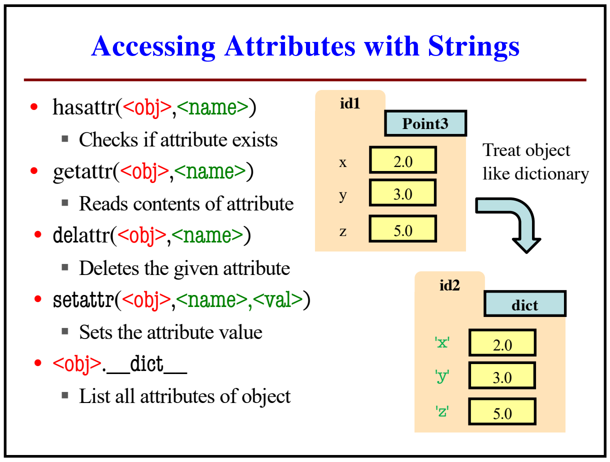


Handling Errors by Type

- try-except can put the error in a variable
- **Example**:

```
try:
    val = input()          # get number from user
    x = float(val)         # convert string to float
    print('The next number is '+str(x+1))
except ValueError as e:
    print(e.args[0])
    print('Hey! That is not a number!')
```

Some Error subclasses have more attributes

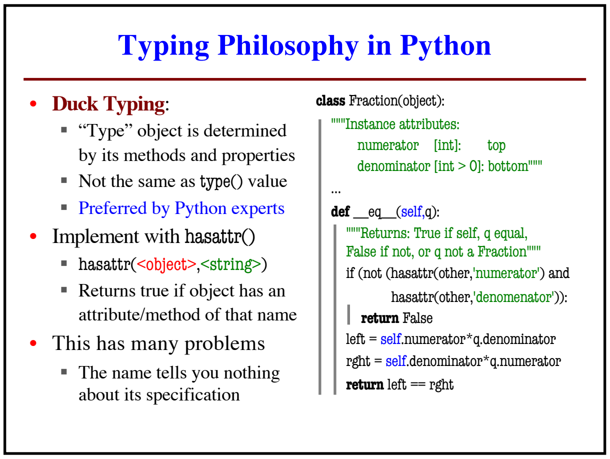


Accessing Attributes with Strings

- hasattr(<obj>,<name>)
  - Checks if attribute exists
- getattr(<obj>,<name>)
  - Reads contents of attribute
- delattr(<obj>,<name>)
  - Deletes the given attribute
- setattr(<obj>,<name>,<val>)
  - Sets the attribute value
- <obj>.__dict__
  - List all attributes of object

id1
Point3
x 2.0
y 3.0
z 5.0
id2
dict
'x' 2.0
'y' 3.0
'z' 5.0

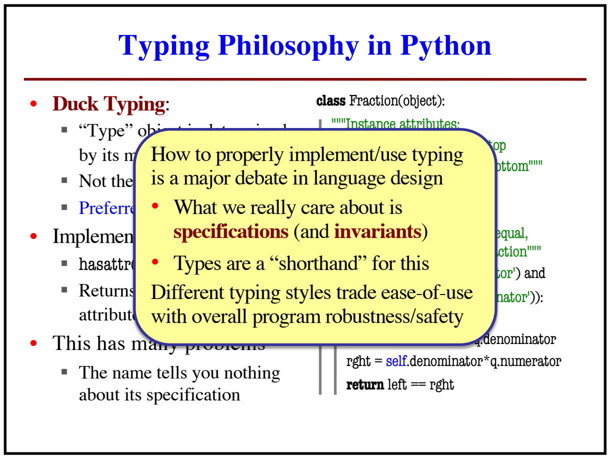
Treat object like dictionary



Typing Philosophy in Python

- **Duck Typing**:
  - "Type" object is determined by its methods and properties
  - Not the same as type() value
  - Preferred by Python experts
- Implement with hasattr()
  - hasattr(<object>,<string>)
  - Returns true if object has an attribute/method of that name
- This has many problems
  - The name tells you nothing about its specification

```
class Fraction(object):
    """Instance attributes:
        numerator    [int]:        top
        denominator [int > 0]: bottom"""
    ...
    def __eq__(self,q):
        """Returns: True if self, q equal,
        False if not, or q not a Fraction"""
        if (not (hasattr(other,'numerator') and
                  hasattr(other,'denomenator')):
            return False
        left = self.numerator*q.denominator
        rght = self.denominator*q.numerator
        return left == rght
```



Typing Philosophy in Python

- **Duck Typing**:
  - "Type" ob
    by its m
  - Not the
  - Preferre
- Implemen
  - hasattr(
  - Returns
    attribut
- This has ma
  - The name tells you nothing about its specification

```
class Fraction(object):
    """Instance attributes:
                                    top
                                    ottom"""

                                    equal,
                                    ction"""
                                    or') and
                                    nator')):

                                    q.denominator
        rght = self.denominator*q.numerator
        return left == rght
```

How to properly implement/use typing is a major debate in language design

- What we really care about is **specifications** (and **invariants**)
- Types are a "shorthand" for this

Different typing styles trade ease-of-use with overall program robustness/safety