61A LECTURE 13 – GENERIC FUNCTIONS

Steven Tang and Eric Tzeng

July 15, 2013

Announcements

- Project 3 released! Due August 2nd.
  - “Plants vs Zombies”!
  - Start early; Project 4 will be the most time consuming project, and we will release it before project 3 is due
- We’ve posted the midterm rubric up on the web page.
  - If you think you got a question COMPLETELY right, but it was marked as wrong, send us an e-mail directly at cs61a@imail.eecs.berkeley.edu
  - If you think one of your questions deserves more partial credit based on the rubric we posted, fill out the midterm 1 regrade request form posted on Piazza. These requests are handled at the end of the semester.

Recapping the midterm

- You guys did well on the midterm!
- Generally, midterms have a mean score of around 35
- This one had a mean of 34, which is about right

Q1 – Proceed with Call-tion

- Average score of 9 / 12
- Important to understand the difference between *returning* a value vs *printing* a value
- Tests your understanding of Python's model of evaluation
- Remember that *or* returns the first value that is True in a Boolean context
  - 25 is returned; not True!
  - Python "short-circuits": 5 / 0 is never evaluated
- Be sure to review this if you did not receive full credit – these types of questions will likely be on future exams

Q2 – Lambda? No thanks.

- 2a had average score of 1.2 / 2
- 2b had average score of 1.5 / 3
- This question tests your understanding of how lambda takes in parameters and returns values
- Common mistake:
  - For 2a, many forgot to include the returned lambda
  - list(x,y) does not construct a list. list() takes exactly one iterable argument. Could do list((x, y)) instead.
  - For 2b, many did not provide ‘zedd’ as an argument to foxes. Also important to index the list at 2, and give the final return call
- These kinds of questions are important; it’s crucial to have a solid grasp of the model of evaluation when learning computer science

Q3 – Tracing through the facts

- 65% of the class got this right
- Essentially an example from lecture 7
- The “traced” version of the function must have the same name as the original, or else the recursive call will not be traced

Q4- Save the environment!

- Average of 3.8 out of 5 for part a,
- Average of 3.7 out of 6 for part b
- Important to remember order of evaluation; lambda's parent is the environment in which the lambda was *evaluated*
- Environment diagrams aren't going anywhere, so be sure to practice them.

Q5 – Testing our potluck

- 74% of the class got part a correct
  - If you had trouble with 5a, come talk to us in office hours.
  - It's important that you understand questions like 5a in the future
- The class average for part b was a 2.7 out of 5
- Essentially, the question required you to loop through a sequence of values, and pick out the highest element
- Also required you to use the correct selectors – you weren't supposed to assume the prediction was a tuple!

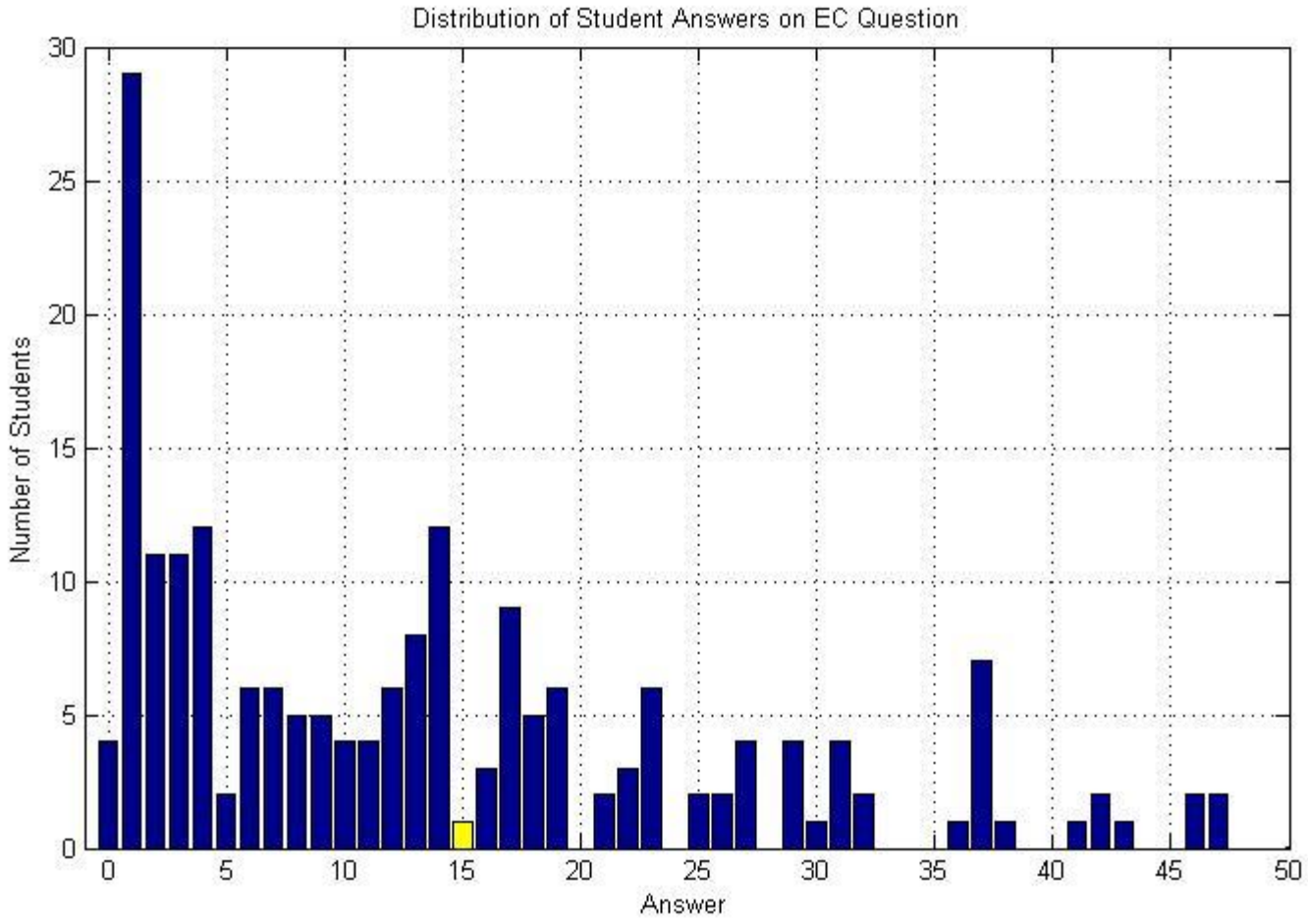
Q6 – Learning to Count

- Part a had an average of 2.1 out of 3
- Part b had an average of 3.2 out of 4
- Part c had an average of 2.8 out of 5
- Note that all of the staff solutions are comfortably 3 lines
- Important to understand what the question is asking for before trying to write any code down

Closing midterm thoughts

- The staff wants everyone to succeed
- Please come to office hours to review concepts that you missed!
- Many of the topics will re-appear in future exams, so it’s crucial to review.

Extra credit results



- 29 people answered “1”, there were only 2 “5’s”
- 12 people answered “14”, but only 1 person answered “**15**”!

Hog results!

- Winners:
- 1. Iann Wu and Cameron Irmas
- 2. Xingchun Wang and Yan Li
- 3. Aditya Chopra and Brian Su
- 4. Jesus Garcia and Andy Chen
- 5. Nihir Patel and Justin Abraham

Details about winning strategies will be posted to Piazza later!

Closing out OOP...

Looking Up Names

Name expressions look up names in the environment

<name>

Dot expressions look up names in an object

<expression> . <name>

```
class CheckingAccount(Account):
    withdraw_fee = 1
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + withdraw_fee)
```

Error: `withdraw_fee` not bound in environment

Not all languages work this way

Designing for Inheritance

Don't repeat yourself; use existing implementations.

Attributes that have been overridden are still accessible via class objects.

Look up attributes on instances whenever possible.

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + self.withdraw_fee)
```

Attribute look-up on base class

Preferable alternative to `CheckingAccount.withdraw_fee`

More practice!

- Write a `Collie` class that does pretty much the same thing as the `Dog` class…
- Except when you tell it to `speak()`, it returns `'there is a boy trapped in the well'` instead of `'woof'`
- And when you tell it to `eat()`, it returns `'this food is exquisite'` instead of `None`

```
class Collie(Dog):
    """A heroic breed of dog"""
    def speak(self):
        Dog.speak(self) #RV of 'woof' not used
        #self.hunger += 1
        return 'there is a boy trapped in the well'
```

Why not use this line instead?

OOP Odds and Ends

The **`object`** class is at the root of the inheritance hierarchy

- **`object`** should be given as the base class when no other meaningful base class exists

Class names should be in CamelCase

Error messages can be confusing when calling methods with the wrong number of arguments:

```
>>> eric_account = Account(100)
>>> eric_account.deposit(100, 200)
TypeError: deposit() takes exactly 2 positional arguments (3 given)
```

Compare to partially curried function:

```
>>> add3 = curry(add)(3)
>>> add3(4, 5)
TypeError: op_add expected 2 arguments, got 3
```

What's next?

- Using the object metaphor, we have bundled together the representation of data and the methods used to manipulate that data, allowing for *modular* programs with local state
- Now, let's see how our object system can allow us to **combine** different types of objects **flexibly** in a large program

Break!

Generic Functions

An abstraction might have more than one representation.

- Python has many sequence types: tuples, ranges, lists, etc.

An abstract data type might have multiple implementations.

- Some representations are better suited to some problems

A function might want to operate on multiple data types.

Message passing enables us to accomplish all of the above, as we will see today and next time

String Representations

An object value should **behave** like the kind of data it is meant to represent;

For instance, by **producing a string** representation of itself.

Strings are important: they represent *language* and *programs*.

In Python, all objects produce two string representations:

- The "str" is legible to **humans**.
- The "repr" is legible to the **Python interpreter**.

"str" and "repr" strings are often the same! Think: numbers.

When the "str" and "repr" **strings are the same**, that's evidence that a programming language is legible by humans!

The “repr” String for an Object

The **`repr`** function returns a Python expression (as a string) that evaluates to an equal object.

```
repr(object) -> string
```

Return the canonical string representation of the object.
For most object types, **`eval(repr(object)) == object`**.

The result of calling **`repr`** on the value of an expression is what Python prints in an interactive session.

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

Some objects don't have a simple Python-readable string.

```
>>> repr(min)
'<built-in function min>'
```

The “str” String for an Object

Human interpretable strings are useful as well:

```
>>> import datetime
>>> today = datetime.date(2013, 7, 16)
>>> repr(today)
'datetime.date(2013, 7, 16)'
>>> str(today)
'2013-07-16'
```

The result of calling **`str`** on the value of an expression is what Python prints using the **`print`** function.

Message Passing Enables Polymorphism

Polymorphic function: A function that can be applied to many (*poly*) different forms (*morph*) of data

`str` and **`repr`** are both polymorphic; they apply to anything.

`repr` invokes a zero-argument method **`__repr__`** on its argument.

```
>>> today.__repr__()
'datetime.date(2013, 7, 16)'
```

`str` invokes a zero-argument method **`__str__`** on its argument. (But **`str`** is a class, not a function!)

```
>>> today.__str__()
'2013-07-16'
```

Inheritance and Polymorphism

Inheritance also enables polymorphism, since subclasses provide at least as much behavior as their base classes

Example of function that works on all accounts:

```
def welfare(account):
    """Deposit $100 into an account if it has less
    than $100."""
    if account.balance < 100:
        return account.deposit(100)
```

```
>>> alice_account = CheckingAccount(0)
>>> welfare(alice_account)
100
>>> bob_account = SavingsAccount(0)
>>> welfare(bob_account)
98
```

Interfaces

Message passing allows **different data types** to respond to the **same message**.

A shared message that elicits similar behavior from different object classes is a powerful method of abstraction.

An *interface* is a **set of shared messages**, along with a specification of **what they mean**.

Classes that implement **`__repr__`** and **`__str__`** methods *that return Python- and human-readable* strings thereby **implement an interface** for producing Python string representations.

Classes that implement **`__len__`** and **`__getitem__`** are sequences.

Special Methods

Python operators and generic functions make use of methods with names like "**`__name__`**"

These are *special* or *magic methods*

Examples:

`len`	**`__len__`**
`+, +=`	**`__add__, __iadd__`**
`[], []=`	**`__getitem__, __setitem__`**
`.`	**`__getattr__, __setattr__`**

Example: Rational Numbers

```
class Rational(object):
    def __init__(self, numer, denom):
        g = gcd(numer, denom)
        self.numerator = numer // g
        self.denominator = denom // g
    def __repr__(self):
        return 'Rational({0}, {1})'.format(self.numerator,
                                           self.denominator)
    def __str__(self):
        return '{0}/{1}'.format(self.numerator,
                                self.denominator)
    def __add__(self, num):
        denom = self.denominator * num.denominator
        numer1 = self.numerator * num.denominator
        numer2 = self.denominator * num.numerator
        return Rational(numer1 + numer2, denom)
    def __eq__(self, num):
        return (self.numerator == num.numerator and
                self.denominator == num.denominator)
```

Property Methods

Often, we want the value of instance attributes to be linked.

```
>>> f = Rational(3, 5)
>>> f.float_value
0.6
>>> f.numerator = 4
>>> f.float_value
0.8
>>> f.denominator -= 3
>>> f.float_value
2.0
```

```
@property
def float_value(self):
    return (self.numerator //
            self.denominator)
```

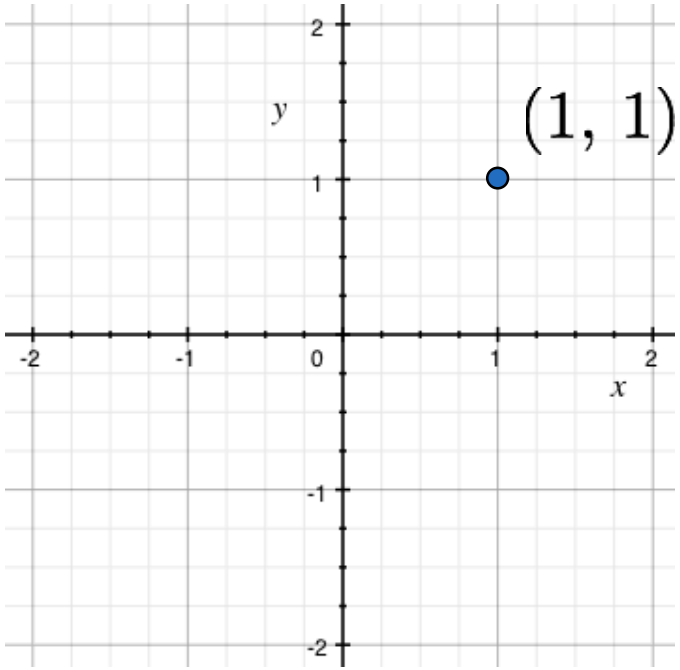
The `@property` decorator on a method designates that it will be called whenever it is *looked up* on an instance.

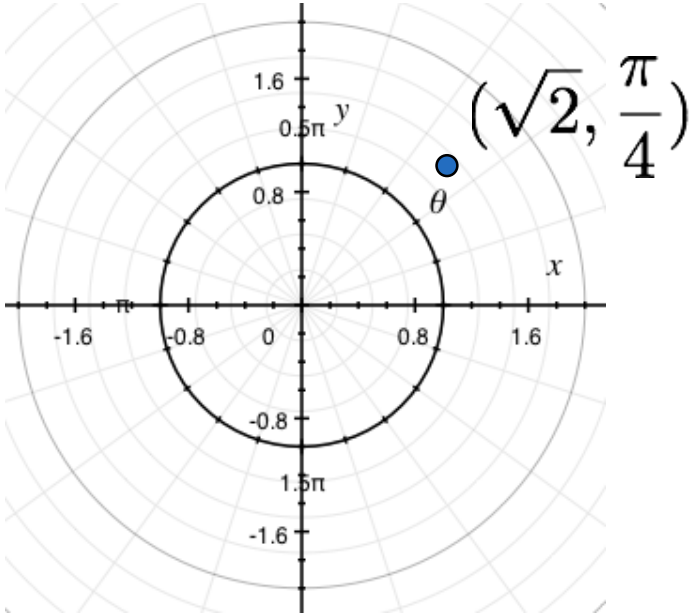
It allows zero-argument methods to be called without an explicit call expression.

Quick Break!

Multiple Representations of Abstract Data

Rectangular and polar representations for complex numbers

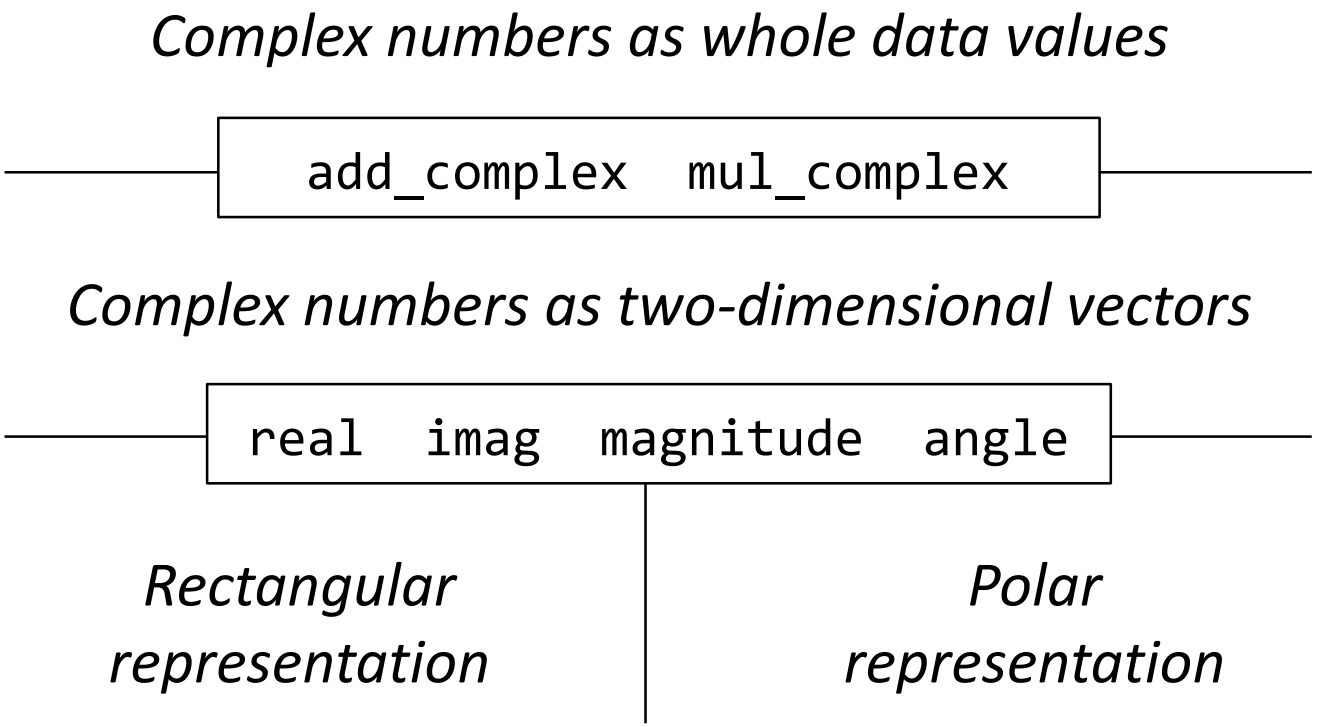




Most operations don't care about the representation.

Some mathematical operations are easier on one than the other.

Arithmetic Abstraction Barriers



An Interface for Complex Numbers

All complex numbers should have real and imag components.

All complex numbers should have a magnitude and angle.

Using this interface, we can implement complex arithmetic:

```
def add_complex(z1, z2):
    return ComplexRI(z1.real + z2.real,
                     z1.imag + z2.imag)

def mul_complex(z1, z2):
    return ComplexMA(z1.magnitude * z2.magnitude,
                     z1.angle + z2.angle)
```

The Rectangular Representation

```
class ComplexRI(object):

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5

    @property
    def angle(self):
        return atan2(self.imag, self.real)

    def __repr__(self):
        return 'ComplexRI({0}, {1})'.format(self.real,
                                            self.imag)
```

Property decorator: "Call this function on attribute look-up"

math.atan2(y,x): Angle between x-axis and the point (x,y)

The Polar Representation

```
class ComplexMA(object):

    def __init__(self, magnitude, angle):
        self.magnitude = magnitude
        self.angle = angle

    @property
    def real(self):
        return self.magnitude * cos(self.angle)

    @property
    def imag(self):
        return self.magnitude * sin(self.angle)

    def __repr__(self):
        return 'ComplexMA({0}, {1})'.format(self.magnitude,
                                            self.angle)
```

Using Complex Numbers

Either type of complex number can be passed as either argument to **`add_complex`** or **`mul_complex`**:

```
def add_complex(z1, z2):
    return ComplexRI(z1.real + z2.real,
                     z1.imag + z2.imag)

def mul_complex(z1, z2):
    return ComplexMA(z1.magnitude * z2.magnitude,
                     z1.angle + z2.angle)
```

```
>>> from math import pi
>>> add_complex(ComplexRI(1, 2), ComplexMA(2, pi/2))
ComplexRI(1.0000000000000002, 4.0)
>>> mul_complex(ComplexRI(0, 1), ComplexRI(0, 1))
ComplexMA(1.0, 3.141592653589793)
```

We can also define **`__add__`** and **`__mul__`** in both classes.