HKN CS61A Midterm 2 Review Session

Alex, Allen, James, Kelvin

Disclaimer

We're not officially affiliated with the course and we don't know what's on your midterm.

This is our best guess based on the lectures, homeworks, past exams, etc.

Always go with the lecture/class material if you feel there is a discrepancy.

Thanks!

Agenda

- Generators and Comprehensions
- OOP
- Abstraction
- Recursion

Not covered:

- Basic usage of built-in types (tuples, strings, dicts, etc)
- Implementation Details

Generators and List Comprehension

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hats(x):
        return x == 42
    def faces(f):
        nonlocal func
        func2 = lambda x: x*2
        it2 = [5, 6, 7, 8]
        if f == "carl":
            func = func2
            it = it2

    if (f == "carl" or f == "paul"):
        faces(f)
        return [func(item) for item in it]
    else:
        return tuple((func(item) for item in it if hats(item)))
```

What does running each of the following tests output?

```
def test1():
    return llamas("carl")
>>> test1()
```

```
def test2():
    return llamas("paul")
>>> test2()
```

```
def test3():
    return llamas("I like hats")
>>> test3()
```

Generators and List Comprehension

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hats(x):
        return x == 42
    def faces(f):
        nonlocal func
        func2 = lambda x: x*2
        it2 = [5, 6, 7, 8]
        if f == "carl":
            func = func2
            it = it2

    if (f == "carl" or f == "paul"):
        faces(f)
        return [func(item) for item in it]
    else:
        return tuple((func(item) for item in it if hats(item)))
```

What does running each of the following tests output?

```
def test1():
    return llamas("carl")
>>> test1()
```

```
def test2():
    return llamas("paul")
>>> test2()
```

```
def test3():
    return llamas("I like hats")
>>> test3()
```

Generators and List Comprehension

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hats(x):
        return x == 42
    def faces(f):
        nonlocal func
        func2 = lambda x: x*2
        it2 = [5, 6, 7, 8]
        if f == "carl":
            func = func2
            it = it2

    if (f == "carl" or f == "paul"):
        faces(f)
        return [func(item) for item in it]
    else:
        return tuple((func(item) for item in it if hats(item)))
```

What does running each of the following tests output?

```
def test1():
    return llamas("carl")
>>> test1()
[(1, 2, 1, 2), (2, 3, 2, 3), (3, 4, 3, 4), (4, 5, 4, 5)]
```

```
def test2():
    return llamas("paul")
>>> test2()
```

```
def test3():
    return llamas("I like hats")
>>> test3()
```

Generators and List Comprehension

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hats(x):
        return x == 42
    def faces(f):
        nonlocal func
        func2 = lambda x: x*2
        it2 = [5, 6, 7, 8]
        if f == "carl":
            func = func2
            it = it2

    if (f == "carl" or f == "paul"):
        faces(f)
        return [func(item) for item in it]
    else:
        return tuple((func(item) for item in it if hats(item)))
```

What does running each of the following tests output?

```
def test1():
    return llamas("carl")
>>> test1()
[(1, 2, 1, 2), (2, 3, 2, 3), (3, 4, 3, 4), (4, 5, 4, 5)]
```

```
def test2():
    return llamas("paul")
>>> test2()
[(1, 2), (2, 3), (3, 4), (4, 5)]
```

```
def test3():
    return llamas("I like hats")
>>> test3()
```

Generators and List Comprehension

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hats(x):
        return x == 42
    def faces(f):
        nonlocal func
        func2 = lambda x: x*2
        it2 = [5, 6, 7, 8]
        if f == "carl":
            func = func2
            it = it2

    if (f == "carl" or f == "paul"):
        faces(f)
        return [func(item) for item in it]
    else:
        return tuple((func(item) for item in it if hats(item)))
```

What does running each of the following tests output?

```
def test1():
    return llamas("carl")
>>> test1()
[(1, 2, 1, 2), (2, 3, 2, 3), (3, 4, 3, 4), (4, 5, 4, 5)]
```

```
def test2():
    return llamas("paul")
>>> test2()
[(1, 2), (2, 3), (3, 4), (4, 5)]
```

```
def test3():
    return llamas("I like hats")
>>> test3()
()
```

Mutability

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hands(h):
        feet = h
        feet[0] = (99, 99)

    if (f == "random man"):
        hands(it)
        return [func(item) for item in it]
```

```
def test4():
    return llamas("random man")
>>> test4()
```

Mutability

```
def llamas(f):
    func = lambda x: x
    it = [(1, 2), (2, 3), (3, 4), (4, 5)]

    def hands(h):
        feet = h
        feet[0] = (99, 99)

    if (f == "random man"):
        hands(it)
        return [func(item) for item in it]
```

```
def test4():
    return llamas("random man")
>>> test4()
[(99, 99), (2, 3), (3, 4), (4, 5)]
```

Map/Reduce/Filter

Given the following, write lines that accomplish each task without loops or defining new functions using only map/reduce/filter and lambda functions

from functools import reduce

>>>line = "raw faces are just gross"

>>>censorlist = ["raw"]

1) Reverses line

'ssorg tsuj era secaf war'

2) Removes words from line that are in the censorlist

'faces are just gross '

3) Doubles each letter then appends "cats" to the front

'cats rraaww ffaacceess aarree jjuusstt ggrroossss'

Map/Reduce/Filter

Given the following, write lines that accomplish each task without loops or defining new functions using only map/reduce/filter and lambda functions

from functools import reduce

>>>line = "raw faces are just gross"

>>>censorlist = ["raw"]

1) Reverses line

>>>str(reduce(lambda x, y: y + x, line))

2) Removes words from line that are in the censorlist

3) Doubles each letter then appends "cats" to the front

Map/Reduce/Filter

Given the following, write lines that accomplish each task without loops or defining new functions using only map/reduce/filter and lambda functions

from functools import reduce

>>>line = "raw faces are just gross"

>>>censorlist = ["raw"]

1) Reverses line

```
>>>str(reduce(lambda x, y: y + x, line))
```

2) Removes words from line that are in the censorlist

```
>>>str(reduce(lambda x, y: x + y, map(lambda x: x + " ", filter(lambda x: x not in censorlist, line.split(" ")))))
```

3) Doubles each letter then appends "cats" to the front

Map/Reduce/Filter

Given the following, write lines that accomplish each task without loops or defining new functions using only map/reduce/filter and lambda functions

from functools import reduce

>>>line = "raw faces are just gross"

>>>censorlist = ["raw"]

1) Reverses line

```
>>>str(reduce(lambda x, y: y + x, line))
```

2) Removes words from line that are in the censorlist

```
>>>str(reduce(lambda x, y: x + y, map(lambda x: x + " ", filter(lambda x: x not in censorlist, line.split(" ")))))
```

3) Doubles each letter then appends "cats" to the front

```
>>>str(reduce(lambda x, y: x + y, map(lambda x: x + x, line), "cats "))
```

OOP Problems

The class **Pokemon** has an attribute `modifiers`:

```
class Pokemon(object):
    modifiers = {"Fire":  1.0,
                 "Water": 1.0,
                 "Grass": 1.0,
                 ... } # Each type gets mapped to 1.0
```

A new class, **PoisonType**, inherits from **Pokemon** but has a few different modifiers (shown on the left)

We want to implement part of the **PoisonType** class such that it has these new modifiers instead of the ones from **Pokemon**.

OOP Problems

Select all of the following that give PoisonType the correct `modifiers` attribute:

```
>>> PoisonType.modifiers["Bug"]
0.5
```

```
>>> Pokemon.modifiers["Bug"]
1.0
```

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
    # (b)
    modifiers = dict(list(Pokemon.modifiers) + list(poison_modifiers))
    # (c)
    modifiers = Pokemon.modifiers.copy()
    modifiers.update(poison_modifiers)
    # (d)
    def modifiers(self, type):
        if type in poison_modifiers.keys():
            return poison_modifiers[type]
        else:
            return modifiers[type]
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
```

Doesn't work!

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
        >>> modifiers
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
        >>> modifiers
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
        >>> modifiers
        *crickets*

        update() doesn't return anything, it just modifies the dict
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (a)
    modifiers = Pokemon.modifiers.update(poison_modifiers)
```

```
>>> modifiers


>>> Pokemon.modifiers
{"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,"Ground":2,
 "Psychic":2,"Fire":1, ... }

Pokemon.modifiers is the dictionary that gets changed!
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (b)
    modifiers = dict( list( Pokemon.modifiers ) + list( poison_modifiers )
    )
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (b)
    modifiers = dict( list( Pokemon.modifiers ) + list( poison_modifiers )
    )
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (b)
    modifiers = dict( list( Pokemon.modifiers ) + list( poison_modifiers )
    )
```

```
>>> list( Pokemon.modifiers )
["Fire", "Water", "Grass", ...]
```

OOP Problems

Possible Solutions (?)

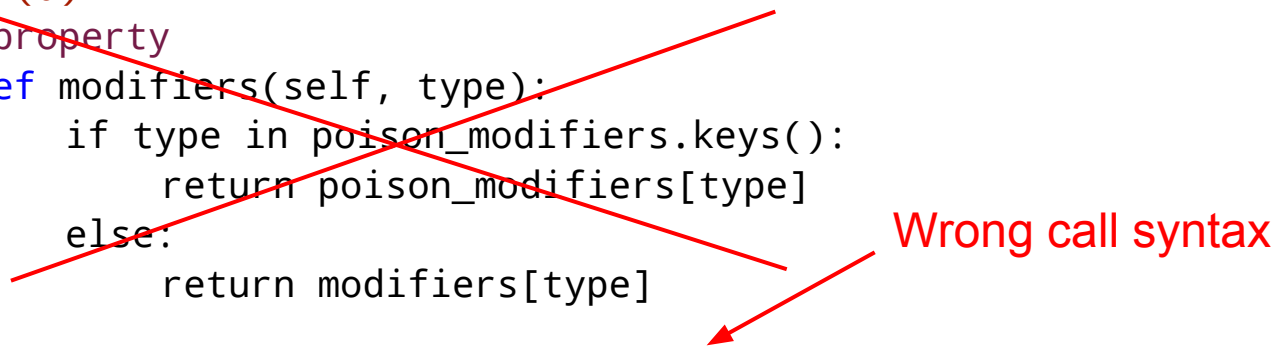
```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (d)
    @property
    def modifiers(self, type):
        if type in poison_modifiers.keys():
            return poison_modifiers[type]
        else:
            return modifiers[type]
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (d)
    @property
    def modifiers(self, type):
        if type in poison_modifiers.keys():
            return poison_modifiers[type]
        else:
            return modifiers[type]

        >>> PoisonType.modifiers["Grass"]
```



OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (d)
    @property
    def modifiers(self, type):
        if type in poison_modifiers.keys():
            return poison_modifiers[type]
        else:
            return modifiers[type]

        >>> PoisonType.modifiers["Grass"]
```

Square brackets call the __getitem__( ) special function,

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (c)
    modifiers = Pokemon.modifiers.copy()
    modifiers.update(poison_modifiers)
```

OOP Problems

Possible Solutions (?)

```
class PoisonType(Pokemon):
    poison_modifiers = {"Bug":0.5,"Fighting":0.5,"Poison":0.5,"Grass":0.5,
    "Ground":2,"Psychic":2}
    # (c)
    modifiers = Pokemon.modifiers.copy()
    modifiers.update(poison_modifiers)
```

THIS VERSION WORKS!

OOP Problems

Assume we have the following classes implemented:

```
class GrassType(Pokemon):
    modifiers =
        Pokemon.modifiers.copy()
    modifiers.update(
        { "Electric": 0.5,
          "Grass":    0.5,
          "Ground":   0.5,
          "Water":    0.5,
          "Bug":      2,
          "Flying":   2,
          "Fire":     2,
          "Ice":      2,
          "Poison":   2} )
```

```
class PoisonType(Pokemon):
    modifiers =
        Pokemon.modifiers.copy()
    modifiers.update(
        { "Bug":      0.5,
          "Fighting": 0.5,
          "Poison":   0.5,
          "Grass":    0.5,
          "Ground":   2,
          "Psychic":  2} )
```

Bulbasaur is a Grass/Poison type pokemon. Suppose we have a new class, **Bulbasaur**, that inherits from these two as follows:

```
class Bulbasaur(GrassType, PoisonType):
```

OOP Problems

```
class GrassType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Electric":   0.5,
            "Grass":      0.5,
            "Ground":     0.5,
            "Water":      0.5,
            "Bug":        2,
            "Flying":     2,
            "Fire":       2,
            "Ice":        2,
            "Poison":     2} )
class PoisonType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Bug":      0.5,
            "Fighting": 0.5,
            "Poison":   0.5,
            "Grass":    0.5,
            "Ground":   2,
            "Psychic":  2} )

class Bulbasaur(GrassType, PoisonType):
```

What do the following statements evaluate to?

```
>>> Bulbasaur.modifiers["Grass"]

>>> Bulbasaur.modifiers["Bug"]

>>> Bulbasaur.modifiers["Psychic"]

>>> Bulbasaur.modifiers["Dragon"]
```

OOP Problems

```
class GrassType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Electric":   0.5,
            "Grass":      0.5,
            "Ground":     0.5,
            "Water":      0.5,
            "Bug":        2,
            "Flying":     2,
            "Fire":       2,
            "Ice":        2,
            "Poison":     2} )
class PoisonType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Bug":      0.5,
            "Fighting": 0.5,
            "Poison":   0.5,
            "Grass":    0.5,
            "Ground":   2,
            "Psychic":  2} )

class Bulbasaur(GrassType, PoisonType):
```

What do the following statements evaluate to?

```
>>> Bulbasaur.modifiers["Grass"]
0.5

>>> Bulbasaur.modifiers["Bug"]

>>> Bulbasaur.modifiers["Psychic"]

>>> Bulbasaur.modifiers["Dragon"]
```

OOP Problems

```
class GrassType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Electric":   0.5,
            "Grass":      0.5,
            "Ground":     0.5,
            "Water":      0.5,
            "Bug":        2,
            "Flying":     2,
            "Fire":       2,
            "Ice":        2,
            "Poison":     2} )
class PoisonType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Bug":      0.5,
            "Fighting": 0.5,
            "Poison":   0.5,
            "Grass":    0.5,
            "Ground":   2,
            "Psychic":  2} )

class Bulbasaur(GrassType, PoisonType):
```

What do the following statements evaluate to?

```
>>> Bulbasaur.modifiers["Grass"]
0.5

>>> Bulbasaur.modifiers["Bug"]
2.0

>>> Bulbasaur.modifiers["Psychic"]

>>> Bulbasaur.modifiers["Dragon"]
```

OOP Problems

```
class GrassType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Electric":   0.5,
            "Grass":      0.5,
            "Ground":     0.5,
            "Water":      0.5,
            "Bug":        2,
            "Flying":     2,
            "Fire":       2,
            "Ice":        2,
            "Poison":     2} )
class PoisonType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Bug":      0.5,
            "Fighting": 0.5,
            "Poison":   0.5,
            "Grass":    0.5,
            "Ground":   2,
            "Psychic":  2} )

class Bulbasaur(GrassType, PoisonType):
```

What do the following statements evaluate to?

```
>>> Bulbasaur.modifiers["Grass"]
0.5

>>> Bulbasaur.modifiers["Bug"]
2.0

>>> Bulbasaur.modifiers["Psychic"]
1.0

>>> Bulbasaur.modifiers["Dragon"]
```

OOP Problems

```
class GrassType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Electric":   0.5,
            "Grass":      0.5,
            "Ground":     0.5,
            "Water":      0.5,
            "Bug":        2,
            "Flying":     2,
            "Fire":       2,
            "Ice":        2,
            "Poison":     2} )
class PoisonType(Pokemon):
     modifiers =
          Pokemon.modifiers.copy()
     modifiers.update(
          { "Bug":      0.5,
            "Fighting": 0.5,
            "Poison":   0.5,
            "Grass":    0.5,
            "Ground":   2,
            "Psychic":  2} )

class Bulbasaur(GrassType, PoisonType):
```

What do the following statements evaluate to?

```
>>> Bulbasaur.modifiers["Grass"]
0.5

>>> Bulbasaur.modifiers["Bug"]
2.0

>>> Bulbasaur.modifiers["Psychic"]
1.0

>>> Bulbasaur.modifiers["Dragon"]
1.0
```

OOP Problems

PROBLEM - Bulbasaur's actual type modifiers don't match *either* type!

Type effectiveness

Under normal battle conditions in Generation V, this Pokémon is:

Damaged normally by:	Normal	1×	Poison	1×	Ground	1×	Rock	1×	Bug	1×	Ghost	1×
	Steel	1×	Dragon	1×	Dark	1×						
Weak to:	Flying	2×	Fire	2×	Psychic	2×	Ice	2×				
Immune to:	None											
Resistant to:	Fighting	½×	Water	½×	Grass	¼×	Electric	½×				

Notes:

- In Generation I, the effectiveness of Bug-type moves is 4×.

SOLUTION - Multiply the modifiers for each type to get the combined modifier for that type.

Write a function that takes two Pokemon types to generate new modifiers.

OOP Problems

```
  GrassType
{ "Electric":0.5,
  "Grass":      0.5,
  "Ground":  0.5,
  "Water":      0.5,
  "Bug":     2.0,
  "Flying":  2.0,
  "Fire":    2.0,
  "Ice":     2.0,
  "Poison":  2.0,
  "Fighting":1.0,
  "Psychic": 1.0,
  ... }
```

```
  PoisonType
{ "Electric": 1.0,
  "Grass":    0.5,
  "Ground":   2.0,
  "Water":    1.0,
  "Bug":      0.5,
  "Flying":   1.0,
  "Fire":     1.0,
  "Ice":      1.0,
  "Poison":   0.5,
  "Fighting": 0.5,
  "Psychic":  2.0,
  ... }
```

```
  Bulbasaur
{ "Electric": 0.5,
  "Grass":    0.25,
  "Ground":   1.0,
  "Water":    0.5,
  "Bug":      1.0,
  "Flying":   2.0,
  "Fire":     2.0,
  "Ice":      2.0,
  "Poison":   1.0,
  "Fighting": 0.5,
  "Psychic":  2.0,
  ... }
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    >>> Bulbasaur.modifiers = combine_types(GrassType, PoisonType)
    >>> GrassType.modifiers["Bug"]
    2.0
    >>> PoisonType.modifiers["Bug"]
    0.5
    >>> Bulbasaur.modifiers["Bug"]
    1.0
    >>> Bulbasaur.modifiers["Flying"]
    2.0
    >>> Bulbasaur.modifiers["Grass"]
    0.25
    """
```

OOP Problems

BONUS:

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    >>> Bulbasaur.modifiers = combine_types(GrassType, PoisonType)
    >>> GrassType.modifiers["Bug"]
    2.0
    >>> PoisonType.modifiers["Bug"]
    0.5
    >>> Bulbasaur.modifiers["Bug"]
    1.0
    >>> Bulbasaur.modifiers["Flying"]
    2.0
    >>> Bulbasaur.modifiers["Grass"]
    0.25
    """
```

OOP Problems

BONUS:

Do it in one line

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    >>> Bulbasaur.modifiers = combine_types(GrassType, PoisonType)
    >>> GrassType.modifiers["Bug"]
    2.0
    >>> PoisonType.modifiers["Bug"]
    0.5
    >>> Bulbasaur.modifiers["Bug"]
    1.0
    >>> Bulbasaur.modifiers["Flying"]
    2.0
    >>> Bulbasaur.modifiers["Grass"]
    0.25
    """
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
    for type in TypeA.modifiers.keys():
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
    for type in TypeA.modifiers.keys():
```

```
["Grass", "Electric", "Water", "Bug", ...]
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
    for type in TypeA.modifiers.keys():
        new_modifiers[type] = TypeA.modifiers[type] * TypeB.modifiers[type]
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
    for type in TypeA.modifiers.keys():
        new_modifiers[type] = TypeA.modifiers[type] * TypeB.modifiers[type]
                    "Poison"              2.0     *         0.5
```

OOP Problems

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    new_modifiers = {}
    for type in TypeA.modifiers.keys():
        new_modifiers[type] = TypeA.modifiers[type] * TypeB.modifiers[type]
    return new_modifiers
```

OOP Problems

BONUS:

Do it in one line

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
```

OOP Problems

BONUS:

Do it in one line

```
def combine_types( TypeA, TypeB ):
    """Combines two different types to form a new type dictionary of modifiers
    """
    return dict( (type, TypeA.modifiers[type] * TypeB.modifiers[type])
                 for type in TypeA.modifiers.keys() )
```

Recursion

```
1  class RList(object):
2      class EmptyList(object):
3          def __len__(self):
4              return 0
5      empty = EmptyList()
6
7      def __init__(self, first, rest=empty):
8          self.first = first
9          self.rest = rest
10     def __len__(self):
11         return 1 + len(self.rest)
12
13
14 blah = RList(1,RList(2,RList(3)))
15
16 def mystery(rlist):
17     if (len(rlist) == 0):
18         return
19     mystery(rlist.rest)
20     print(rlist.first)
21
22 mystery(blah)
```

Given this mystery function and the corresponding RList object, what does this mystery function do?

What would Python print after calling mystery on blah?

Recursion

```
1  class RList(object):
2      class EmptyList(object):
3          def __len__(self):
4              return 0
5      empty = EmptyList()
6
7      def __init__(self, first, rest=empty):
8          self.first = first
9          self.rest = rest
10     def __len__(self):
11         return 1 + len(self.rest)
12
13
14 blah = RList(1,RList(2,RList(3)))
15
16 def mystery(rlist):
17     if (len(rlist) == 0):
18         return
19     mystery(rlist.rest)
20     print(rlist.first)
21
22 mystery(blah)
```

Given this mystery function and the corresponding RList object, what does this mystery function do?

Print out what's contained in the RList backwards.

What would Python print after calling mystery on blah?

3

2

1

Recursion

Given a list of integers, without using any for or while loops, write a function to count the instances of odd numbers.

Recursion

Given a list of integers, without using any for or while loops, write a function to count the instances of odd numbers.

```
24 def odds(nums):
25     if (len(nums) == 0):
26         return 0
27     return nums[0] % 2 + odds(nums[1:])
```

Recursion

```
38 def factorial(n):
39     if (n == 0):
40         return 1
41     return n * factorial(n-1)
42 
43 def sum_factorial(n):
44     sum = 0
45     for i in range(0,n+1):
46         sum += factorial(i)
47     return sum
```

Given this factorial function what is the Big O of the sum_factorial function?

Recursion

```
38 def factorial(n):
39     if (n == 0):
40         return 1
41     return n * factorial(n-1)
42 
43 def sum_factorial(n):
44     sum = 0
45     for i in range(0,n+1):
46         sum += factorial(i)
47     return sum
```

Given this function, what is the Big O of this function?

O(n) = n^2

Recursion

```
38 def factorial(n):
39     if (n == 0):
40         return 1
41     return n * factorial(n-1)
42 
43 def sum_factorial(n):
44     sum = 0
45     for i in range(0,n+1):
46         sum += factorial(i)
47     return sum
```

Using this original function, implement the memoized function that was implemented in class. After implementing it, figure out the Big O of the memoized sum_factorial.

Recursion

```
29 def memo(f):
30     cache = {}
31     def memoized(n):
32         if n not in cache:
33             cache[n] = f(n)
34         return cache[n]
35     return memoized
36
37 @memo
38 def factorial(n):
39     if (n == 0):
40         return 1
41     return n * factorial(n-1)
42
43 def sum_factorial(n):
44     sum = 0
45     for i in range(0,n+1):
46         sum += factorial(i)
47     return sum
```

The way to memoize is on the left.

Big O is now

O(n) = n

Since, you only have to calculate each factorial once, and if you only add one more, its constant time.

Trees

```
class Tree(object):
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

Given the class tree object that is shown on the left. Write a function that will sum all the values that are within this tree.

Trees

```
class Tree(object):
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

```
def sumValues(tree):
    if (tree == None):
        return 0
    return tree.value + sumValues(tree.left) + sumValues(tree.right)
```

Trees

```
class Tree(object):
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

```
def sumValues(tree):
    if (tree == None):
        return 0
    return tree.value + sumValues(tree.left) + sumValues(tree.right)
```

What is the Big O of this function, given that n is number of nodes?

Trees

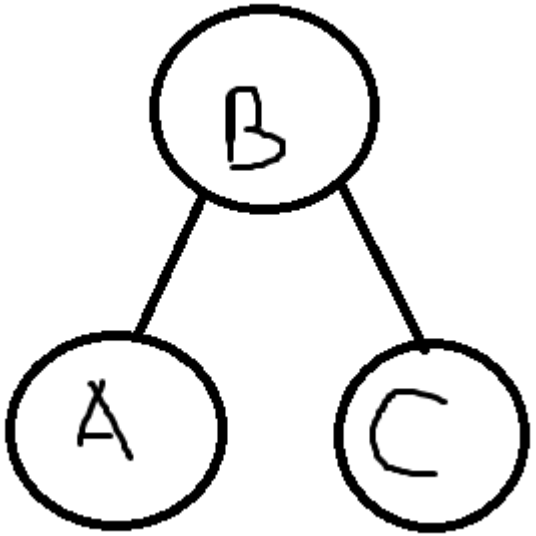
```
class Tree(object):
    def __init__(self, value, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

```
def sumValues(tree):
    if (tree == None):
        return 0
    return tree.value + sumValues(tree.left) + sumValues(tree.right)
```

What is the Big O of this function, given that n is number of nodes?

O(n) = n

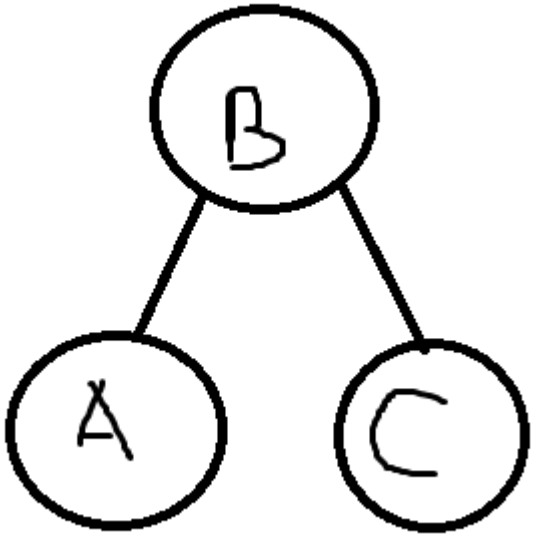
Trees




```
64 def inorder(tree):
65     if(tree == None):
66         return
67     inorder(tree.left)
68     print(tree.value)
69     inorder(tree.right)
70
```

This is the function for an inorder traversal of a tree, what would this function print given this tree?

Trees




```
64 def inorder(tree):
65     if(tree == None):
66         return
67     inorder(tree.left)
68     print(tree.value)
69     inorder(tree.right)
70
```

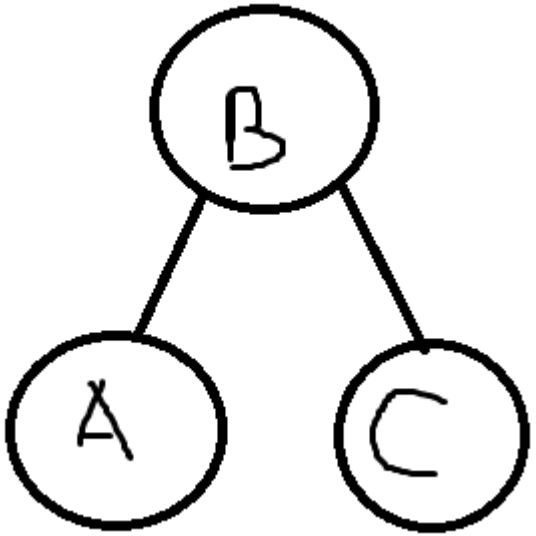
This is the function for an inorder traversal of a tree, what would this function print given this tree?

A

B

C

Trees



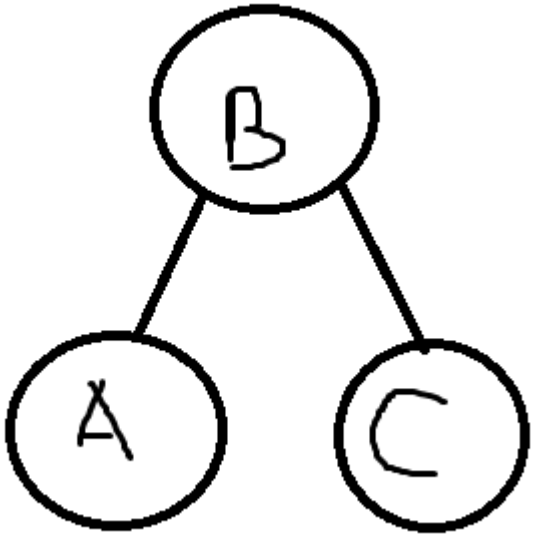

```
64 def inorder(tree):
65     if(tree == None):
66         return
67     inorder(tree.left)
68     print(tree.value)
69     inorder(tree.right)
70
```

Now you know how an inorder traversal works, write the Preorder Traversal and the Postorder Traversal.

Preorder:B, A, C

Postorder:A, C, B

Trees




```
64 def inorder(tree):
65     if(tree == None):
66         return
67     inorder(tree.left)
68     print(tree.value)
69     inorder(tree.right)
70
```

```
71 def preorder(tree):
72     if(tree == None):
73         return
74     print (tree.value)
75     preorder(tree.left)
76     preorder(tree.right)
77
78 def postorder(tree)
79     if(tree == None):
80         return
81     postorder(tree.left)
82     postorder(tree.right)
83     print(tree.value)
```

Tree

```
64 def inorder(tree):
65     if(tree == None):
66         return
67     inorder(tree.left)
68     print(tree.value)
69     inorder(tree.right)
70
71 def preorder(tree):
72     if(tree == None):
73         return
74     print (tree.value)
75     preorder(tree.left)
76     preorder(tree.right)
77
78 def postorder(tree)
79     if(tree == None):
80         return
81     postorder(tree.left)
82     postorder(tree.right)
83     print(tree.value)
```

Now that we have these functions, lets build a tree using what you know from here.

Inorder:

A, B, C, D, E, F, G, H

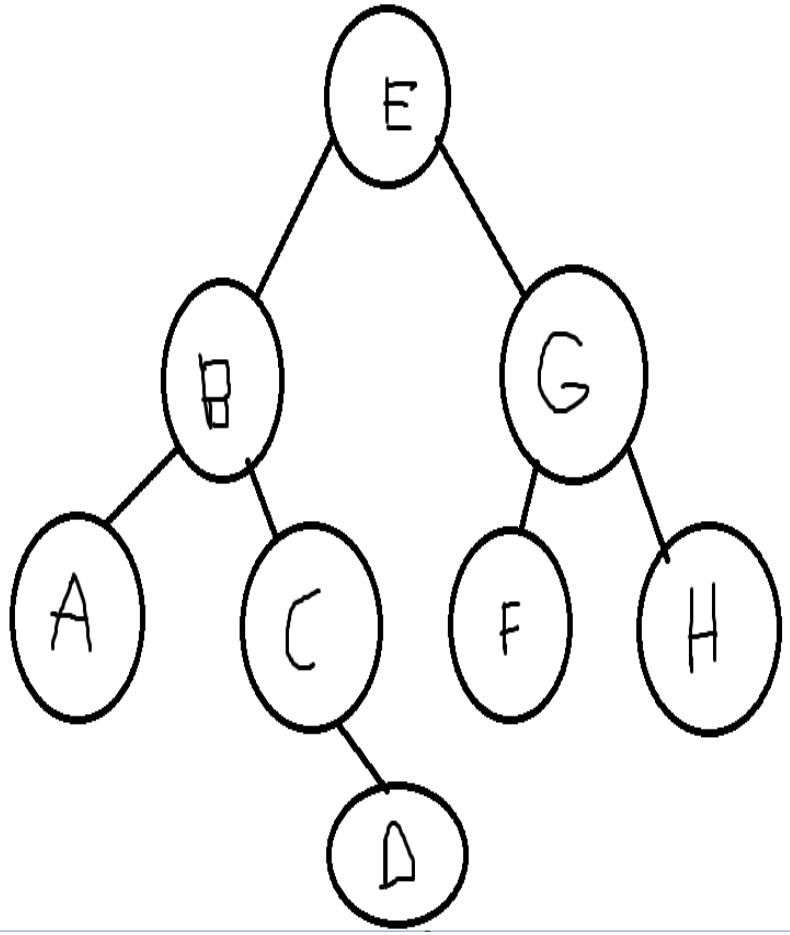
Preorder:

E, B, A, C, D, G, F, H

Postorder:

A, D, C, B, F, H, G, E

Tree



Now that we have these functions, lets build a tree using what you know from here.

Inorder:

A, B, C, D, E, F, G, H

Preorder:

E, B, A, C, D, G, F, H

Postorder:

A, D, C, B, F, H, G, E

Interfaces

An interface describes a set of messages and their meaning.

Let's define the "catchable" interface as these methods:

catch(self, owner) :
assigns the owner of this catchable to be "owner", but only if original owner is "None"
if the owner isn't "None", call the owner's block_pokeball()

release(self) :
assigns the owner of this catchable to be "None"

owner(self) :
returns the owner of this catchable

in particular, we want any "catchable" pokemon to be able to work with something like:

```
def trade(catchable1, catchable2):
    owner1 = catchable1.owner()
    owner2 = catchable2.owner()
    catchable1.release()
    catchable1.catch(owner2)
    catchable2.release()
    catchable2.catch(owner1)
```

Interfaces

Does this class correctly implement "catchable"?

```
class Missingno(Pokemon):

    < ... other class attributes, methods, etc >

    def catch(self, owner):
        if self.__owner != None:
            self.__owner.block_pokeball()
        elif owner != None:
            self.__owner = owner
            self.__owner.corrupt_save_data()

    def release(self):
        if self.__owner != None:
            self.__owner.corrupt_save_data()
        else:
            self.__owner = None

    def owner(self):
        return self.__owner
```

catch(self, owner) :

assigns the owner of this catchable to be "owner", but only if old owner is "None"
if the owner isn't "None", call the owner's block_pokeball()

release(self) :

assigns the owner of this catchable to be "None"

owner(self) :

returns the owner of this catchable

Interfaces

Does this class correctly implement "catchable"?

```
class Missingno(Pokemon):

   < ... other class attributes, methods, etc >

   def catch(self, owner):
      if self.__owner != None:
         self.__owner.block_pokeball()
      elif owner != None:
         self.__owner = owner
         self.__owner.corrupt_save_data()

   def release(self):
      if self.__owner != None:
         self.__owner.corrupt_save_data()
      else:
         self.__owner = None

   def owner(self):
      return self.__owner
```

catch(self, owner) :
assigns the owner of this catchable to be "owner", but only if old owner is "None"
if the owner isn't "None", call the owner's block_pokeball()

release(self) :
assigns the owner of this catchable to be "None"

owner(self) :
returns the owner of this catchable

No.

Obvious reason:

release() fails to set owner to None if it isn't already

Less obvious:

corrupt_save_data() might have side effects that don't fit the interface, but it depends on what corrupt_save_data() does and what the interface intends/allows.

__repr__ and __str__

- A common Python **interface** for getting a string representation.
- __repr__ should produce something legible to the interpreter and unambiguous to the identity of the object.
- __str__ should produce something a human can read and understand.

Multiple Representations

Data can be represented in many different forms

you've seen strings, tuples, lists, ... even functions

That one company is designing a contacts system for the Pokegear/Pokenav/Poketech/C-Gear/Xtransciever. Specifically, they want an Entry type which stores an id, name, and number.

Here's an implementation that does it all using strings (terrible choice).

No id/name/number ever contains a tab character.

```
class Entry:
    def __init__(self, id, name, number):
        self.__data = str(id) + '\t' + str(name) + '\t' + str(number)
    def get(self, key):
        if key is 'id':
            return int(self.__data.split('\t')[0])
        elif key is 'name':
            return str(self.__data.split('\t')[1])
        elif key is 'number':
            return int(self.__data.split('\t')[2])
```

Using a different representation of the data, implement Entry (ours is 5 lines total).

Multiple Representations

Using a different representation of the data, implement Entry (ours is 5 lines total).

```
class Entry:
    def __init__(self, id, name, number):
        self.__data = {'id':id, 'name':name, 'number':number}
    def get(self, key):
        return self.__data[key]
```

Other ways include: store as fields, store in a list, store in a tuple ...

Type dispatching and Data-directed Programming

Type dispatching:

- We can use a different function for each possible combination of types on an operator and using the correct function for a provided combination of types.
- In lecture, you explored tag-based table method.

Data-directed Programming:

- a dispatching function performs table lookups with the operator and types to apply the correct function.

Type Coercion

Sometimes, it makes sense to do an operation dispatched between different types by converting one of the types into another.
The way to do this is obvious with many numeric types (an integer can be coerced to a rational number can be coerced to a real number).
It is not always the case that one type is a subset of the other.

We have count_legendaries(ls), a function which takes in a list and outputs the number of legendaries in the list. For example:

```
>>> count_legendaries(['Unown', 'Geodude', 'Mewtwo'])
1
```

Make a coercion so that we can count the legendaries in comma-separated strings, such as 'Bulbasaur,Charmander,Squirtle' using count_legendaries.

```
def cs_to_ls(comma_separated_string):
    return _____________________________
```

Type Coercion

Sometimes, it makes sense to do an operation dispatched between different types by converting one of the types into another.
The way to do this is obvious with many numeric types (an integer can be coerced to a rational number can be coerced to a real number).
It is not always the case that one type is a subset of the other.

We have count_legendaries(ls), a function which takes in a list and outputs the number of legendaries in the list. For example:

```
>>> count_legendaries(['Unown', 'Geodude', 'Mewtwo'])
1
```

Make a coercion so that we can count the legendaries in comma-separated strings, such as 'Bulbasaur,Charmander,Squirtle' using count_legendaries.

```
def cs_to_ls(comma_separated_string):
    return comma_separated_string.split(',')
```