CS 61A Midterm 2 Study Guide — Page 1

Numeric types in Python:

```
>>> type(2)
<class 'int'>
```
Represents integers exactly

```
>>> type(1.5)
<class 'float'>
```
Represents real numbers approximately

```
>>> type(1+1j)
<class 'complex'>
```

User-defined complex type:

```
>>> z = ComplexRI(-1, 0)
>>> z.real
-1
>>> z.magnitude
1
>>> z.angle
3.141592653589793
```

```
class ComplexRI:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag
    @property
    def magnitude(self):
        return (self.real
    @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)

Type dispatching: Look up a cross-type implementation of an operation based on the types of its arguments
Data-directed programming: Look up a cross-type implementation based on both the operator and types of its arguments
Type coercion: Look up a function for converting one type to another, then apply a type-specific implementation.

Rational number: $\dfrac{\text{numerator}}{\text{denominator}}$

- Exact representation of fractions
- A pair of integers
- As soon as division occurs, the exact representation may be lost!
- Assume we can compose and decompose rational numbers:

Constructor
- rational(n, d) *returns a rational number* x

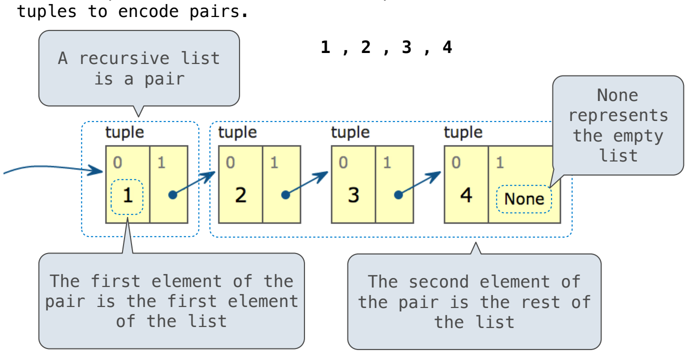
Selectors
- numer(x) *returns the numerator of* x
- denom(x) *returns the denominator of* x

These functions implement an *abstract data type* for rational numbers

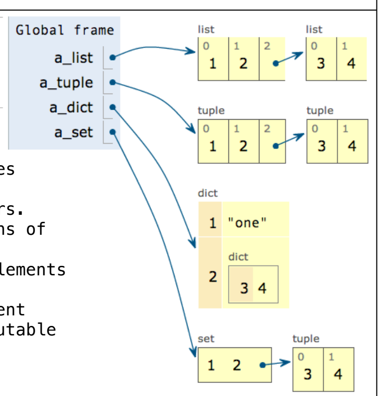
There isn't just one sequence class or abstract data type (in Python or in general).
The sequence abstraction is a collection of behaviors:

Length. A sequence has a finite length.
Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

We can implement recursive lists as pairs. We'll use two-element tuples to encode pairs.




```
a_list = [1, 2, [3, 4]]
a_tuple = (1, 2, (3, 4))
a_dict = {1: 'one', 2: {3: 4}}
a_set = {1, 2, (3, 4)}
```



- Lists are mutable sequences
- Tuples are immutable sequences
- Dictionaries are unordered collections of key-value pairs.
- Sets are unordered collections of values.
- Two dictionary keys or set elements cannot be equal.
- A dictionary key or set element cannot be an instance of a mutable built-in type.

Executing a for statement:

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the first frame of the current environment.
   B. Execute the <suite>.

A range is a sequence of consecutive integers.

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

range(-2, 2)

Length: ending value - starting value
Element selection: starting value + index

```
>>> tuple(range(-2, 2))
(-2, -1, 0, 1)
>>> tuple(range(4))
(0, 1, 2, 3)
```

Tuple constructor

With a 0 starting value

List comprehensions: [<map exp> for <name> in <iter exp> if <filter exp>]
Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list by this procedure:
1. Add a new frame extending the current frame.
2. Create an empty *result list* that is the value of the expression.
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1.
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list.

Strings are sequences too:

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string, but with only one character!

The "in" and "not in" operators match substrings

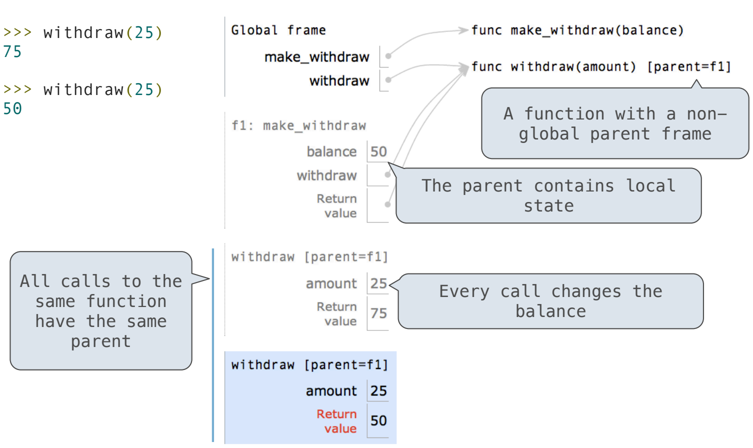
```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in (1, 2, 3, 4, 5)
False
```

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

Re-bind balance in the first non-local frame in which it was bound previously

```
>>> withdraw(25)
75
>>> withdraw(25)
50
```



Status x = 2	**Effect**
•No nonlocal statement •"x" **is not** bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment.
•No nonlocal statement •"x" **is** bound locally	Re-bind name "x" to object 2 in the first frame of the current env.
•nonlocal x •"x" **is** bound in a non-local frame	Re-bind "x" to 2 in the first non-local frame of the current environment in which it is bound.
•nonlocal x •"x" **is not** bound in a non-local frame	SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •"x" **is** bound in a non-local frame •"x" also bound locally	SyntaxError: name 'x' is parameter and nonlocal

```
class <name>:
    <suite>
```

The suite is executed when a class statement is evaluated.

A class statement **creates** a new class and **binds** that class to <name> in the first frame of the current environment.

Statements in the <suite> create attributes of the class.

As soon as an instance is created, it is passed to __init__, which is a class attribute called the *constructor method*.

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance **or** its class.

<expression> . <name>

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute **looked up** by <name> in the object that is the value of the <expression>.

```
tom_account.deposit(10)
```

Dot expression — Call expression

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression.
2. <name> is matched against the instance attributes of that object; **if an attribute with that name exists**, its value is returned.
3. If not, <name> is looked up in the class, which yields a class attribute value.
4. That value is returned **unless it is a function**, in which case a *bound method* is returned instead.

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- For an instance, then assignment sets an instance attribute
- For a class, then assignment sets a class attribute

Account class attributes: interest: ~~0.02~~ ~~0.04~~ (withdraw, deposit, __init__)

Instance attributes of jim_account: balance: 0, holder: 'Jim', interest: 0.08

Instance attributes of tom_account: balance: 0, holder: 'Tom'

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```

Inheritance: To look up a name in a class.
1. If it names an attribute in the class, return its value.
2. Otherwise, look up the name in the base class, if it exists.

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

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)

>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest      # Found in CheckingAccount
0.01
>>> ch.deposit(20)   # Found in Account
20
>>> ch.withdraw(5)   # Found in CheckingAccount
14

class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1           # A free dollar!
```

When a class is called:

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
```

1. A new instance of that class is created: { balance: 0, holder: 'Jim' }
2. The constructor __init__ of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity testing is performed by "is" and "is not" operators:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment **does not** create a new object:

```
>>> c = a
>>> c is a
True
```

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

```
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two arguments

Dot notation automatically supplies the first argument to a method.

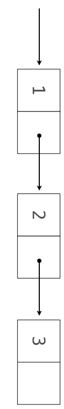
```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Invoked with one argument

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest
    def __getitem__(self, index):
        if index == 0:
            return self.first
        else:
            return self.rest[index-1]
    def __len__(self):
        return 1 + len(self.rest)
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    else:
        return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

There's the base case!

Yes, this call is recursive

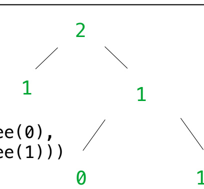


Rlist(1, Rlist(2, Rlist(3)))

```
class Tree:
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
def sum_entries(t):
    if t is None:
        return 0
    else:
        return t.entry + sum_entries(t.left) + sum_entries(t.right)
```

left and right are Trees or None

```
Tree(2, Tree(1),
        Tree(1, Tree(0),
                Tree(1)))
```




```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

n: size of the problem

R(n): Measurement of some resource

$$R(n) = \Theta(f(n))$$

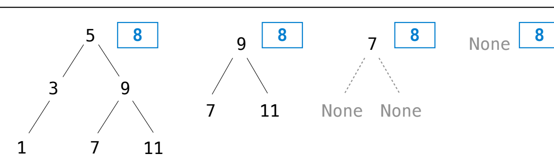
means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

- $\Theta(b^n)$ Exponential growth! Incrementing the problem scales R(n) by a factor.
- $\Theta(n^2)$ Quadratic growth. Incrementing n increases R(n) by the problem size n.
- $\Theta(n)$ Linear growth. Resources scale with the problem size.
- $\Theta(\log n)$ Logarithmic growth. Doubling the problem only increments R(n).
- $\Theta(1)$ Constant. Independent of problem size.

Tree set: A set is a Tree. Each entry is:
- Larger than all entries in its left branch, and
- Smaller than all entries in its right branch



Right! *Left!* *Right!* *Stop!*