

Numeric types in Python:

```
>>> type(2)
<class 'int'>
```

Represents integers exactly

```
>>> type(1.5)
<class 'float'>
```

Represents real numbers with finite precision

Rational implementation using functions:

```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select
```

This function represents a rational number

```
def numer(x):
    return x('n')
```

Constructor is a higher-order function

```
def denom(x):
    return x('d')
```

Selector calls x

Lists:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```

digits

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

list

list

Executing a for statement:

- ```
for <name> in <expression>:
 <suite>
```
1. Evaluate the header **<expression>**, which must yield an iterable value (a sequence)
  2. For each element in that sequence, in order:
    - A. Bind **<name>** to that element in the current frame
    - B. Execute the **<suite>**

## Unpacking in a for statement:

A sequence of fixed-length sequences

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0
```

A name for each element in a fixed-length sequence

```
>>> for x, y in pairs:
... if x == y:
... same_count = same_count + 1
>>> same_count
2
```

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

range(-2, 2)

**Length:** ending value – starting value

**Element selection:** starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]
```

List constructor

```
>>> list(range(4))
[0, 1, 2, 3]
```

Range with a 0 starting value

## Membership:

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

## Slicing:

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

## Functions that aggregate iterable arguments

```
• sum(iterable[, start]) -> value
• max(iterable[, key=func]) -> value
 max(a, b, c, ..., [key=func]) -> value
• min(iterable[, key=func]) -> value
 min(a, b, c, ..., [key=func]) -> value
• all(iterable) -> bool
• any(iterable) -> bool
```

## 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 using this evaluation procedure:

1. Add a new frame with the current frame as its parent
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

The result of calling **repr** on a value is what Python prints in an interactive session

The result of calling **str** on a value is what Python prints using the **print** function

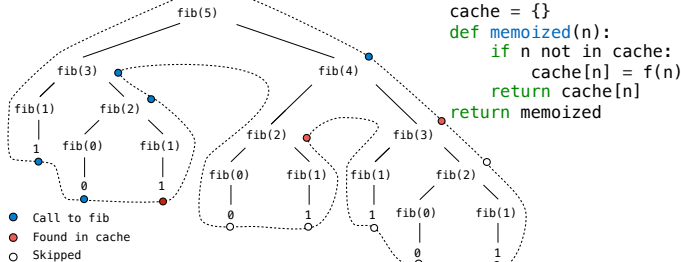
```
>>> 12e12
12000000000000.0
>>> print(12e12)
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

**str** and **repr** are both polymorphic; they apply to any object

**repr** invokes a zero-argument method **\_\_repr\_\_** on its argument

```
>>> today.__repr__()
'datetime.date(2014, 10, 13)'
>>> today.__str__()
'2014-10-13'
```

## Memoization:



**Type dispatching:** Look up a cross-type implementation of an operation based on the types of its arguments

**Type coercion:** Look up a function for converting one type to another, then apply a type-specific implementation.

$R(n) = \Theta(f(n))$  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 all  $n$  larger than some  $m$

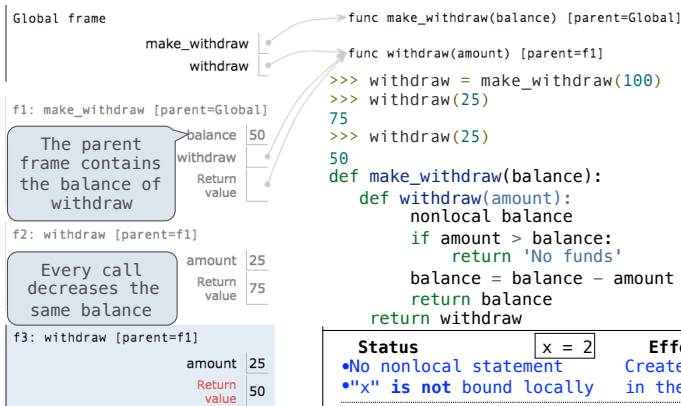
$\Theta(b^n)$  Exponential growth. Recursive **fib** takes  $\Theta(\phi^n)$  steps, where  $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$ . Incrementing the problem scales  $R(n)$  by a factor  $\phi$

$\Theta(n^2)$  Quadratic growth. E.g., **overlap**. Incrementing  $n$  increases  $R(n)$  by the problem size  $n$

$\Theta(n)$  Linear growth. E.g., **factors** or **exp**

$\Theta(\log n)$  Logarithmic growth. E.g., **exp\_fast**. Doubling the problem only increments  $R(n)$

$\Theta(1)$  Constant. The problem size doesn't matter



## Strings as sequences:

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4]
False
```

## List &amp; dictionary mutation:

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
True
```

```
>>> nums = {'I': 1.0, 'V': 5, 'X': 10}
>>> nums['X']
10
>>> nums['I'] = 1
>>> nums['L'] = 50
>>> nums
{'X': 10, 'L': 50, 'V': 5, 'I': 1}
>>> sum(nums.values())
66
>>> dict([(3, 9), (4, 16), (5, 25)])
{3: 9, 4: 16, 5: 25}
>>> nums.get('A', 0)
0
>>> nums.get('V', 0)
5
>>> {x: x*x for x in range(3,6)}
{3: 9, 4: 16, 5: 25}
```

```
>>> suits = ['coin', 'string', 'myriad']
>>> suits.pop()
'myriad'
>>> suits.remove('string')
>>> suits.append('cup')
>>> suits.extend(['sword', 'club'])
>>> suits[2] = 'spade'
>>> suits
['coin', 'cup', 'spade', 'club']
>>> suits[0:2] = ['diamond']
>>> suits
['diamond', 'spade', 'club']
>>> suits.insert(0, 'heart')
>>> suits
['heart', 'diamond', 'spade', 'club']
```

Remove and return the last element

Remove a value

Add all values

Replace a slice with values

Add an element at an index

## Identity:

**<exp0> is <exp1>** evaluates to **True** if both **<exp0>** and **<exp1>** evaluate to the same object

**<exp0> == <exp1>** evaluates to **True** if both **<exp0>** and **<exp1>** evaluate to equal values

**Identical objects are always equal values**

You can **copy** a list by calling the list constructor or slicing the list from the beginning to the end.

**Constants:** Constant terms do not affect the order of growth of a process

$\Theta(n)$   $\Theta(500 \cdot n)$   $\Theta(\frac{n}{500})$

**Logarithms:** The base of a logarithm does not affect the order of growth of a process

$\Theta(\log_2 n)$   $\Theta(\log_{10} n)$   $\Theta(\ln n)$

**Nesting:** When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
 count = 0
 for item in a:
 if item in b:
 count += 1
 return count
```

Outer: length of a

Inner: length of b

If  $a$  and  $b$  are both length  $n$ , then **overlap** takes  $\Theta(n^2)$  steps

**Lower-order terms:** The fastest-growing part of the computation dominates the total

$\Theta(n^2)$   $\Theta(n^2 + n)$   $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

## Status

• No nonlocal statement

• "x" is not bound locally

• No nonlocal statement

• "x" is bound locally

• nonlocal x

• "x" is bound in a non-local frame

• nonlocal x

• "x" is not bound in a non-local frame

• nonlocal x

• "x" is bound in a non-local frame

• "x" also bound locally

## Effect

Create a new binding from name "x" to number 2 in the first frame of the current environment

Re-bind name "x" to object 2 in the first frame of the current environment

SyntaxError: no binding for nonlocal 'x' found in a non-local frame

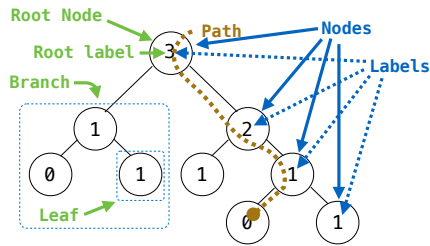
SyntaxError: name 'x' is parameter and nonlocal

**Recursive description:**

- A tree has a root label and a list of branches
- Each branch is a tree
- A tree with zero branches is called a leaf

**Relative description:**

- Each location is a node
- Each node has a label
- One node can be the parent/child of another



```
def tree(label, branches=[]):
```

```
 for branch in branches:
```

```
 assert is_tree(branch)
```

```
 return [label] + list(branches)
```

```
def label(tree):
```

```
 return tree[0]
```

```
def branches(tree):
```

```
 return tree[1:]
```

```
def is_tree(tree):
```

```
 if type(tree) != list or len(tree) < 1:
```

```
 return False
```

```
 for branch in branches(tree):
```

```
 if not is_tree(branch):
```

```
 return False
```

```
 return True
```

```
def is_leaf(tree):
```

```
 return not branches(tree)
```

```
def leaves(t):
```

```
 """The leaf values in t.
```

```
 >>> leaves(fib_tree(5))
```

```
 [1, 0, 1, 0, 1, 1, 0, 1]
```

```
 """
```

```
 if is_leaf(t):
```

```
 return [label(t)]
```

```
 else:
```

```
 return sum([leaves(b) for b in branches(t)], [])
```

```
class Tree:
```

```
 def __init__(self, label, branches=[]):
```

```
 self.label = label
```

```
 for branch in branches:
```

```
 assert isinstance(branch, Tree)
```

```
 self.branches = list(branches)
```

```
 def is_leaf(self):
```

```
 return not self.branches
```

```
def leaves(tree):
```

```
 """The leaf values in a tree."
```

```
 if tree.is_leaf():
```

```
 return [tree.label]
```

```
 else:
```

```
 return sum([leaves(b) for b in tree.branches], [])
```

```
class Link:
```

```
 empty = ()
```

```
 def __init__(self, first, rest=empty):
```

```
 assert rest is Link.empty or isinstance(rest, Link)
```

```
 self.first = first
```

```
 self.rest = rest
```

```
 def __repr__(self):
```

```
 if self.rest:
```

```
 rest_str = ' ' + repr(self.rest)
```

```
 else:
```

```
 rest_str = ''
```

```
 return 'Link({0}{1})'.format(self.first,
```

```
 rest_str)
```

```
 def __str__(self):
```

```
 string = '<'
```

```
 while self.rest is not Link.empty:
```

```
 string += str(self.first) + ', '
```

```
 self = self.rest
```

```
 return string + str(self.first) + '>'
```

```
Python built-in sets:
```

```
>>> s = {3, 2, 1, 4, 4}
```

```
>>> s
```

```
{1, 2, 3, 4}
```

```
>>> 3 in s
```

```
True
```

```
>>> len(s)
```

```
4
```

```
>>> s.union({1, 5})
```

```
{1, 2, 3, 4, 5}
```

```
>>> s.intersection({6, 5, 4, 3})
```

```
{3, 4}
```

A binary search tree is a binary tree where each root is larger than all values in its left branch and smaller than all values in its right branch

```
class BTree(Tree):
```

```
 empty = Tree(None)
```

```
 def __init__(self, label, left=empty, right=empty):
```

```
 Tree.__init__(self, label, [left, right])
```

```
@property
```

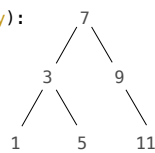
```
 def left(self):
```

```
 return self.branches[0]
```

```
@property
```

```
 def right(self):
```

```
 return self.branches[1]
```

**Python object system:**

**Idea:** All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

```
balance: 0 holder: 'Jim'
```

When a class is called:

1. A new instance of that class is created:

2. The `__init__` method 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.

`__init__` is called a constructor

```
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
```

`self` should always be bound to an instance of the Account class or a subclass of Account

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

Method invocation: One object before the dot and other arguments within parentheses

```
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Call expression

Dot expression

`<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>`.

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

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

Account class attributes

```
interest: 0.02 0.04 0.05
(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
0.04
```

```
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)
 or
 return super().withdraw(
 amount + self.withdraw_fee)
```

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> 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
```