



DM536

Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

<http://imada.sdu.dk/~petersk/DM536/>

LIST PROCESSING

Lists as Sequences

- lists are sequences of values
- lists can be constructed using “[” and “]”
- Example:
 - `[42, 23]`
 - `["Hello", "World", "!"]`
 - `["strings and", int, "mix", 2]`
 - `[]`
- lists can be nested, i.e., a list can contain other lists
- Example: `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
- lists are normal values, i.e., they can be printed, assigned etc.
- Example:
 - `x = [1, 2, 3]`
 - `print x, [x, x], [[x, x], x]`

Mutable Lists

- lists can be accessed using indices
- lists are mutable, i.e., they can be changed destructively
- Example:

```
x = [1, 2, 3]
```

```
print x[1]
```

```
x[1] = 4
```

```
print x, x[1]
```

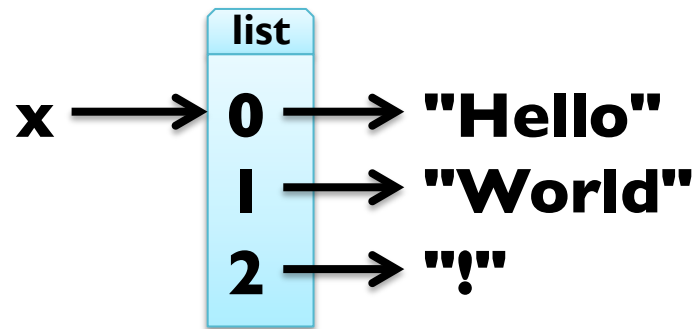
- `len(object)` and negative values work like for strings
- Example:

```
x[2] == x[-1]
```

```
x[1] == x[len(x)-2]
```

Stack Diagrams with Lists

- lists can be viewed as mappings from indices to elements
- Example 1: $x = ["\text{Hello}", "\text{World}", "!"]$



- Example 2: $x = [[23, 42, -3.0], "\text{Bye!}"]$



Traversing Lists

- `for` loop consecutively assigns variable to elements of list
- Example: print squares of numbers from 1 to 10
for `x` in `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`:
 `print x**2`
- arithmetic sequences can be generated using `range` function:
 - `range([start,] stop[, step])`
- Example:
 `range(4) == [0, 1, 2, 3]`
 `range(1, 11) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
 `range(9, 1, -2) == [9, 7, 5, 3]`
 `range(1, 10, 2) == [1, 3, 5, 7, 9]`

Traversing Lists

- `for` loop consecutively assigns variable to elements of list

- general form

```
for element in my_list:  
    print element
```

- iteration through list with indices:

```
for index in range(len(my_list)):  
    element = my_list[index]  
    print element
```

- Example: in-situ update of list

```
x = [8388608, 43980465 | | | 04, 0.125]
```

```
for i in range(len(x)):
```

```
    x[i] = math.log(x[i], 2)
```

List Operations

- like for strings, “+” concatenates two lists

- Example:

$[1, 2, 3] + [4, 5, 6] == \text{range}(1, 7)$

$[[23, 42] + [-3.0]] + ["Bye!"] == [[23, 42, -3.0], "Bye!"]$

- like for strings, “* n” with integer n produces n copies

- Example:

$\text{len}(["I", "love", "penguins!"] * 100) == 300$

$(\text{range}(1, 3) + \text{range}(3, 1, -1)) * 2 == [1, 2, 3, 2, 1, 2, 3, 2]$

List Slices

- slices work just like for strings
- Example:

```
x = ["Hello", 2, "u", 2, "!"]  
x[2:4] == ["u", 2]  
x[2:] == x[-3:len(x)]  
y = x[:]      # make a copy (lists are mutable!)
```
- **BUT:** we can also assign to slices!
- Example:

```
x[1:4] = ["to", "you", "too"]  
x == ["Hello", "to", "you", "too", "!"]  
x[1:3] = ["to me"]  
x == ["Hello", "to me", "too", "!"]  
x[2:3] = []  
x == ["Hello", "to me", "!"]
```

List Methods

- appending elements to the end of the list (destructive)
- Example: `x = [5, 3, 1]`
`y = [2, 4, 6]`
for e in y: `x.append(e)`
- Note: `x += [e]` would create new list in each step!
- also available as method: `x.extend(y)`
- sorting elements in ascending order (destructive)
- Example: `x.sort()`
`x == range(1, 7)`
- careful with destructive updates: `x = x.sort()`

Higher-Order Functions (map)

- Example 1: new list with squares of all elements of a list

```
def square_all(x):
```

```
    res = []
```

```
    for e in x:    res.append(e**2)
```

```
    return res
```

- Example 2: new list with all elements increased by one

```
def increment_all(x):
```

```
    res = []
```

```
    for e in x:    res.append(e+1)
```

```
    return res
```

Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []
```

```
for e in x: res.append(e**2)
```

```
return res
```

```
res = []
```

```
for e in x: res.append(e+1)
```

```
return res
```

- Python has generic function `map(function, sequence)`
- Implementation idea:

```
def map(function, sequence):
```

```
    res = []
```

```
    for e in sequence:
```

```
        res.append(function(e))
```

```
    return res
```

Higher-Order Functions (map)

- these *map* operations have an identical structure:

```
res = []  
for e in x: res.append(e**2)  
return res
```

```
res = []  
for e in x: res.append(e+1)  
return res
```

- Python has generic function `map(function, sequence)`
- Example:

```
def square(x):      return x**2
```

```
def increment(x):   return x+1
```

```
def square_all(x):  
    return map(square, x)
```

```
def increment_all(x):  
    return map(increment, x)
```

Higher-Order Functions (filter)

- Example 1: new list with elements greater than 42

```
def filter_greater42(x):
```

```
    res = []
```

```
    for e in x:
```

```
        if e > 42:    res.append(e)
```

```
    return res
```

- Example 2: new list with elements whose length is smaller 3

```
def filter_len_smaller3(x):
```

```
    res = []
```

```
    for e in x:
```

```
        if len(e) < 3:    res.append(e)
```

```
    return res
```

Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
```

```
for e in x:
```

```
    if e > 42: res.append(e)
```

```
return res
```

```
res = []
```

```
for e in x:
```

```
    if len(e) < 3: res.append(e)
```

```
return res
```

- Python has generic function `filter(function, iterable)`
- Implementation idea:

```
def filter(function, iterable):
```

```
    res = []
```

```
    for e in iterable:
```

```
        if function(e): res.append(e)
```

```
    return res
```

Higher-Order Functions (filter)

- these *filter* operations have an identical structure:

```
res = []
```

```
for e in x:
```

```
    if e > 42: res.append(e)
```

```
return res
```

```
res = []
```

```
for e in x:
```

```
    if len(e) < 3: res.append(e)
```

```
return res
```

- Python has generic function `filter(function, iterable)`
- Example:

```
def greater42(x):
```

```
    return x > 42
```

```
def len_smaller3(x):
```

```
    return len(x) < 3
```

```
def filter_greater42(x):
```

```
    return filter(greater42, x)
```

```
def filter_len_smaller3(x):
```

```
    return filter(len_smaller3, x)
```


Higher-Order Functions (reduce)

- Example 1: computing factorial using range

```
def mul_all(x):
```

```
    prod = 1
```

```
    for e in x:    prod *= e                # prod = prod * e
```

```
    return prod
```

```
def factorial(n):
```

```
    return mul_all(range(1,n+1))
```

- Example 2: summing all elements in a list

```
def add_all(x):
```

```
    sum = 0
```

```
    for e in x:    sum += e                # sum = sum + e
```

```
    return sum
```

Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1
```

```
for e in x: prod *= e
```

```
return prod
```

```
sum = 0
```

```
for e in x: sum += e
```

```
return sum
```

- Python has generic function `reduce(function, sequence, initial)`
- Implementation idea:

```
def reduce(function, sequence, initial):
```

```
    result = initial
```

```
    for e in sequence:
```

```
        result = function(result, e)
```

```
    return result
```

Higher-Order Functions (reduce)

- these *reduce* operations have an identical structure:

```
prod = 1
```

```
for e in x: prod *= e
```

```
return prod
```

```
sum = 0
```

```
for e in x: sum += e
```

```
return sum
```

- Python has generic function `reduce(function, sequence, initial)`
- Example:

```
def add(x,y): return x+y
```

```
def mul(x,y): return x*y
```

```
def add_all(x):
```

```
    return reduce(add, x, 0)
```

```
def mul_all(x):
```

```
    return reduce(mul, x, 1)
```

Deleting Elements

- there are three different ways to delete elements from list
- if you know index and want the element, use `pop(index)`
- Example:

```
my_list = [23, 42, -3.0, 47 | | ]  
my_list.pop(1) == 42  
my_list == [23, -3.0, 47 | | ]
```
- if you do not know index, but the element, use `remove(value)`
- Example:

```
my_list.remove(-3.0)  
my_list == [23, 47 | | ]
```
- if you know the index, you can use the `del` statement
- Example:

```
del my_list[0]  
my_list == [47 | | ]
```

Deleting Elements

- there are three different ways to delete elements from list
- as we have seen, you can also use slices to delete elements
- Example:

```
my_list = [23, 42, -3.0, 47 | | ]  
my_list[2:] = []  
my_list == [23, 42]
```
- alternatively, you can use `del` together with slices
- Example:

```
my_list = my_list * 3  
del my_list[:3]  
my_list == [42, 23, 42]
```

Lists vs Strings

- string = sequence of letters
- list = sequence of values
- convert a string into a list using the built-in `list()` function
- Example: `list("Hej hop") == ["H", "e", "j", " ", "h", "o", "p"]`
- split up a string into a list using the `split(sep)` method
- Example: `"Slartibartfast".split("a") == ["Sl", "rtib", "rtf", "st"]`
- reverse operation is the `join(sequence)` method
- Example: `" and ".join(["A", "B", "C"]) == "A and B and C"`
`"".join(["H", "e", "j", " ", "h", "o", "p"]) = "Hej Hop"`

Objects and Values

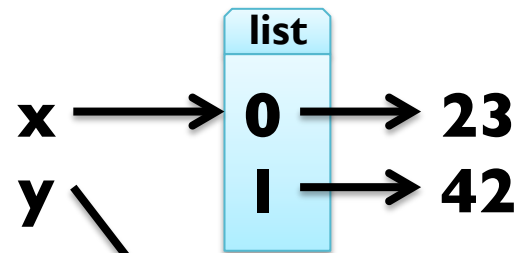
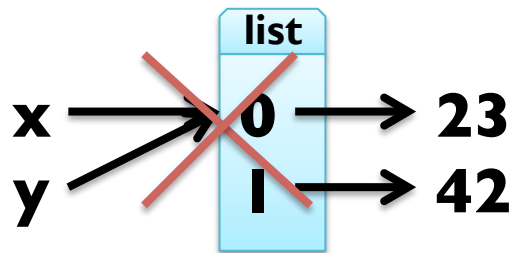
- two possible stack diagrams for `a = "mango"; b = "mango"`



- we can check identity of objects using the `is` operator

- Example: `a is b == True`

- two possible stack diagrams for `x = [23, 42]; y = [23, 42]`



- Example: `x is y == False`

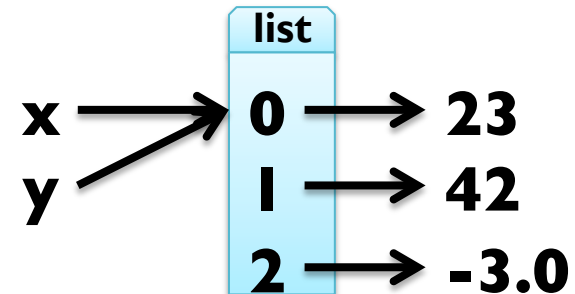
Aliasing

- when assigning $y = x$, both variables refer to same object

- Example: $x = [23, 42, -3.0]$

$y = x$

$x \text{ is } y == \text{True}$



- here, there are two *references* to one (*aliased*) object

- fine for immutable objects (like strings)

- problematic for mutable objects (like lists)

- Example: $y[2] = 4711$

$x == [23, 42, 4711]$

- HINT: when unsure, always copy list using $y = x[:]$

List Arguments

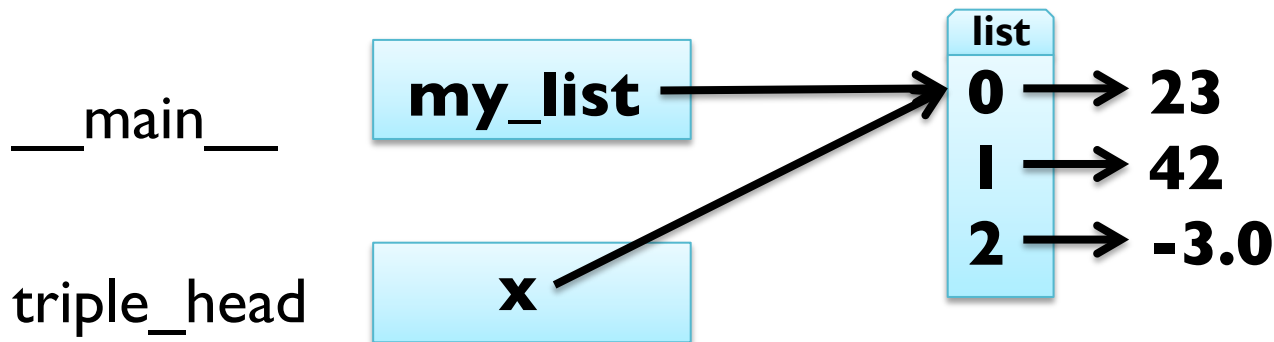
- lists passed as arguments to functions can be changed
- Example: tripling the first element

```
def triple_head(x):
```

```
    x[:1] = [x[0]]*3
```

```
my_list = [23, 42, -3.0]
```

```
triple_head(my_list)
```



List Arguments

- lists passed as arguments to functions can be changed
- Example: tripling the first element

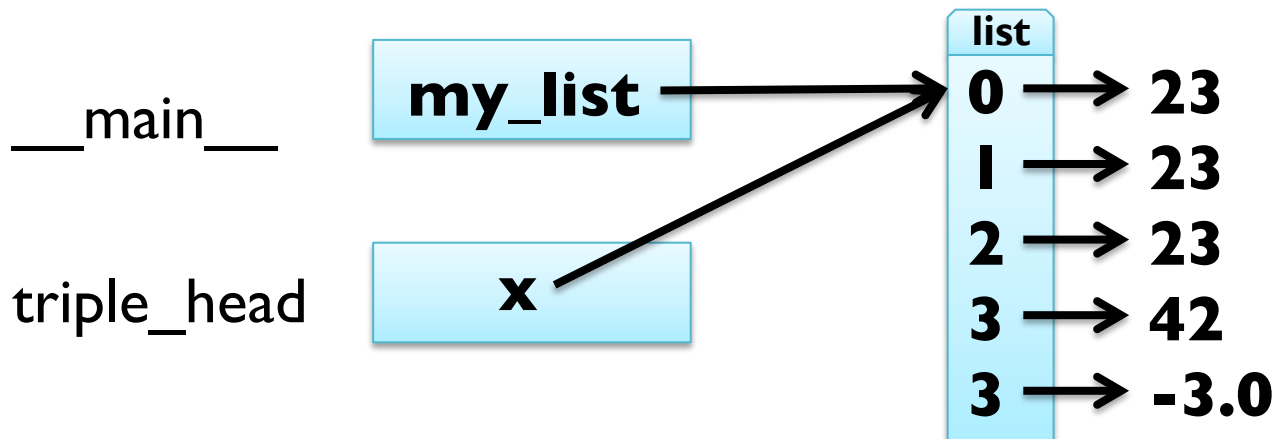
```
def triple_head(x):
```

```
    x[:1] = [x[0]]*3
```

```
my_list = [23, 42, -3.0]
```

```
triple_head(my_list)
```

```
my_list == [23, 23, 23, 42, -3.0]
```



List Arguments

- lists passed as arguments to functions can be changed
- some operations change object
 - assignment using indices
 - `append(object)` method
 - `extend(iterable)` method
 - `sort()` method
 - `del` statement
- some operations return a new object
 - access using slices
 - `strip()` method
 - “+” on strings and lists
 - “* n” on strings and lists

Debugging Lists

- working with mutable objects like lists requires attention!
 1. many list methods return **None** and modify destructively
 - `word = word.strip()` makes sense
 - `t = t.sort()` does **NOT!**
 2. there are many ways to do something – stick with one!
 - `t.append(x)` or `t = t + [x]`
 - use either `pop`, `remove`, `del` or slice assignment for deletion
 3. make copies when you are unsure!
 - Example:

```
...
sorted_list = my_list[:]
sorted_list.sort()
...
```

DICTIONARIES

Generalized Mappings

- list = mapping from integer indices to values
- dictionary = mapping from (almost) any type to values
- indices are called *keys* and pairs of keys and values *items*
- empty dictionaries created using curly braces “{}”
- Example: `en2da = {}`
- keys are assigned to values using same syntax as for sequences
- Example: `en2da["queen"] = "dronning"`
`print en2da`
- curly braces “{” and “}” can be used to create dictionary
- Example: `en2da = {"queen" : "dronning", "king" : "konge"}`

Dictionary Operations

- printing order can be different: `print en2da`
- access using indices: `en2da["king"] == "konge"`
- `KeyError` when key not mapped: `print en2da["prince"]`
- length is number of items: `len(en2da) == 2`
- `in` operator tests if key mapped: `"king" in en2da == True`
`"prince" in en2da == False`
- `keys()` method gives list of keys:
`en2da.keys() == ["king", "queen"]`
- `values()` method gives list of values:
`en2da.values() == ["konge", "dronning"]`
- useful e.g. for test if value is used:
`"prins" in en2da.values() == False`

Dictionaries as Sets

- dictionaries can be used as sets
- **Idea:** assign `None` to all elements of the set
- Example: representing the set of primes smaller than 20
`primes = {2: None, 3: None, 5: None, 7: None, 11: None, 13: None, 17: None, 19: None}`
- then `in` operator can be used to see if value is in set
- Example:
`15 in primes == False`
`17 in primes == True`
- for lists, needs steps proportional to number of elements
- for dictionary, needs (almost) constant number of steps

Counting Letter Frequency

- **Goal:** count frequency of letters in a string (*histogram*)
- many possible implementations, e.g.:
 - create 26(+3?) counter variables for each letter!; use chained conditionals (`if ... elif ... elif ...`) to increment
 - create a list of length 26(+3?); increment the element at index $n-1$ if the n -th letter is encountered
 - create a dictionary with letters as keys and integers as values; increment using index access
- all these implementations work (differently)
- big differences in *runtime* and *ease of implementation*
- choice of data structure is a *design decision*

Counting with Dictionaries

- fast and counts all characters – no need to fix before!

```
def histogram(word):
```

```
    d = {}
```

```
    for char in word:
```

```
        if char not in d:
```

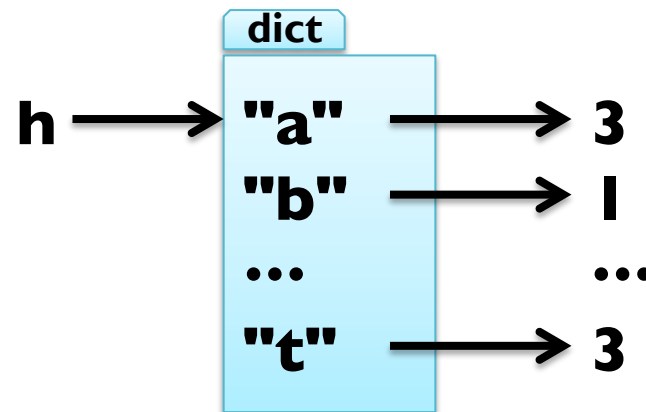
```
            d[char] = 1
```

```
        else:
```

```
            d[char] += 1
```

```
    return d
```

- Example: `h = histogram("slartibartfast")`
`h == {"a":3, "b":1, "f":1, "i":1, "l":1, "s":2, "r":2, "t":3}`



Counting with Dictionaries

- fast and counts all characters – no need to fix before!

```
def histogram(word):
```

```
    d = {}
```

```
    for char in word:
```

```
        if char not in d:
```

```
            d[char] = 1
```

```
        else:
```

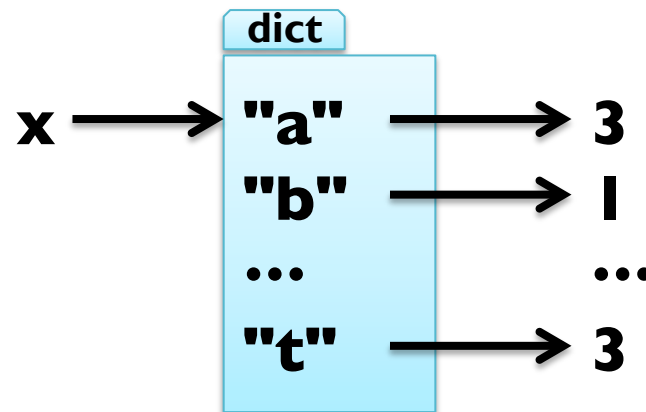
```
            d[char] += 1
```

```
    return d
```

- access using the `get(k, d)` method:

```
h.get("t", 0) == 3
```

```
h.get("z", 0) == 0
```



Traversing Dictionaries

- using a `for` loop, you can traverse all keys of a dictionary

- Example: `for key in en2da:`

```
    print key, en2da[key]
```

- you can also traverse all values of a dictionary

- Example: `for value in en2da.values():`

```
    print value
```

- finally, you can traverse all items of a dictionary

- Example: `for item in en2da.items():`

```
    print item[0], item[1]    # key, value
```

Reverse Lookup

- given dict. `d` and key `k`, finding value `v` with `v == d[k]` easy
- this is called a dictionary *lookup*
- given dict. `d` and value `v`, finding key `k` with `v == d[k]` hard
- there might be more than one key mapping to `v` (cf. example)
- Possible implementation I:

```
def reverse_lookup(d, v):  
    result = []  
    for key in d:  
        if d[key] == v:  
            result.append(key)  
    return result
```

- returns empty list, when no key maps to value `v`

Reverse Lookup

- given dict. d and key k , finding value v with $v == d[k]$ easy
- this is called a dictionary *lookup*
- given dict. d and value v , finding key k with $v == d[k]$ hard
- there might be more than one key mapping to v (cf. example)
- Possible implementation 2:

```
def reverse_lookup(d, v):
```

```
    for k in d:
```

```
        if d[k] == v:
```

```
            return k
```

```
    raise ValueError
```

- gives error when no key maps to value v

Reverse Lookup

- given dict. d and key k , finding value v with $v == d[k]$ easy
- this is called a dictionary *lookup*
- given dict. d and value v , finding key k with $v == d[k]$ hard
- there might be more than one key mapping to v (cf. example)
- Possible implementation 2:

```
def reverse_lookup(d, v):  
    for k in d:  
        if d[k] == v:  
            return k  
    raise ValueError, "value not found in dictionary"
```

- gives error when no key maps to value v

Dictionaries and Lists

- lists cannot be keys, as they are mutable
- list can be values stored in dictionaries
- Example: inverting a dictionary

```
def invert_dict(d):
```

```
    inv = {}
```

```
    for key in d:
```

```
        val = d[key]
```

```
        if val not in inv:
```

```
            inv[val] = [key]
```

```
        else:
```

```
            inv[val].append(key)
```

```
    return inv
```


Dictionaries and Lists

- lists cannot be keys, as they are mutable
- list can be values
- Example: inverting a dictionary

```
def invert_dict(d):
```

```
    inv = {}
```

```
    for key in d:
```

```
        val = d[key]
```

```
        if val not in inv:
```

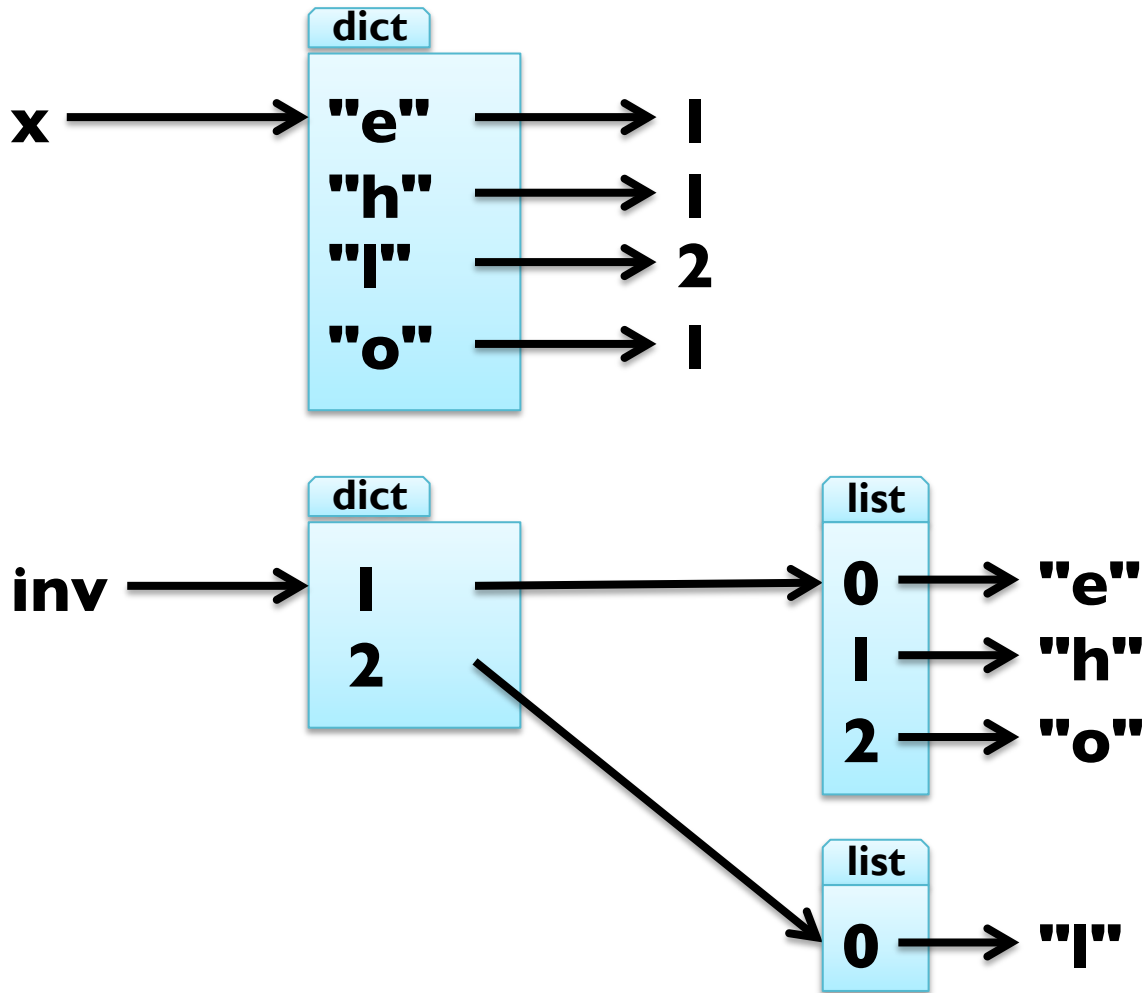
```
            inv[val] = []
```

```
            inv[val].append(key)
```

```
    return inv
```

- Example: `print invert_dict(histogram("hello"))`

Dictionaries and Lists



- Example: `print invert_dict(histogram("hello"))`

Memoizing

- Fibonacci numbers lead to exponentially many calls:

```
def fib(n):
```

```
    if n in [0,1]: return n
```

```
    return fib(n-1) + fib(n-2)
```

- keeping previously computed values (*memos*) helps:

```
known = {0:0, 1:1}
```

```
def fib_fast(n):
```

```
    if n in known:
```

```
        return known[n]
```

```
    res = fib_fast(n-1) + fib_fast(n-2)
```

```
    known[n] = res
```

```
    return res
```

Global Variables

- known is created outside `fib_fast` and belongs to `__main__`
- such variables are called *global*
- many uses for global variables (besides memoization)
- Example 1: flag for controlling output

```
debug = True
```

```
def pythagoras(a,b):
```

```
    if debug:    print "pythagoras with a =d", a, " and b = d", b
```

```
    result = math.sqrt(a**2 + b**2)
```

```
    if debug:    print "result of pythagoras:", result
```

```
    return result
```

Global Variables

- known is created outside `fib_fast` and belongs to `__main__`
- such variables are called *global*
- many uses for global variables (besides memoization)
- Example 2: track number of calls

```
num_calls = 0
```

```
def pythagoras(a,b):
```

```
    global num_calls
```

```
    num_calls += 1
```

```
    return math.sqrt(a**2 + b**2)
```

- gives `UnboundLocalError` as `num_calls` is local to `pythagoras`
- declare `num_calls` to be global using a `global` statement

Long Integers

- Python uses 32 or 64 bit for `int`
- this limits the numbers that can be represented:
 - 32 bit: from -2^{31} to $2^{31}-1$
 - 64 bit: from -2^{63} to $2^{63}-1$
- for larger numbers, Python automatically uses `long` integers
- Example:
`fib(93) == 12200160415121876738L`
- `long` integers work just like `int`, only with "L" as suffix
- Example: `2**64 + 2**64 == 2**65`
`fib(100)**fib(20) # has 139016 digits :-o`

Debugging Larger Datasets

- debugging larger data sets, simple printing can be too much
 1. scale down the input – start with the first n lines; a good value for n is a small value that still exhibits the problem
 2. scale down the output – just print a part of the output; when using strings and lists, slices are very handy
 3. check summaries and types – check that type and len(...) of objects is correct by printing them instead of the object
 4. write self-checks – include some *sanity checks*, i.e., test Boolean conditions that should definitely hold
 5. pretty print output – even larger sets can be easier to interpret when printed in a more human-readable form