

# Python Intermediate Programming

## Unit 0: Introduction

PYTHON REVIEW III COMPOUND DATA TYPES AND AGGREGATE FUNCTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER

# Collections in Python





QlikView

# Aggregate Functions

# Comprehensions: list, tuple, set, dict

LECTURE 1



# Python List Comprehension



1

A Revision of List  
Comprehension

2

Syntax

3

List Comprehension vs  
Lambda Expressions

4

Conditionals in List  
Comprehension

5

Nested Loops in List  
Comprehension



# Comprehensions: list, tuple, set, dict

---

- **Comprehensions** are compact ways to create complicated (but not too complicated) **lists, tuples, sets, and dicts**. That is, they compactly solve some problems but cannot solve all problems (for example, we cannot use them to mutate values in an existing data structure, just to create values in a new data structure).
- The general form of a list comprehension is as follows, where *f* means any function using *var* (or expression using *var*: we can also write just *var* there because a name by itself is a very simple expression) and *p* means any predicate (or bool expression) using *var*.

```
[f(var, ...) for var in iterable if p(var, ...)]
```



# Comprehensions

---

- **Meaning:** collect together into a list (because of the outer `[]`) all of `f(var,...)` values, for `var` taking on every value in iterable, but only collect an `f(var,...)` value if its corresponding `p(var,...)` is `True`.
- For tuple or set comprehensions, we would use `()` and `{}` as the outermost grouping symbol instead of `[]`. We'll talk about dicts at the end, which use `{}` but also with a `:` inside (separating keys from values) to be distinguished from sets, which use `{}` without any such `:` inside.



# Comprehensions

---

- Note that the "if p(var,...)" part is optional, so we can also write the simplest comprehensions as follows (in which case it has the same meaning as p(var,...) always being True).

```
[f(var,...) for var in iterable]
```

which has the same meaning as

```
[f(var,...) for var in iterable if True]
```

for example

```
x = [i**2 for i in irange(1,10) if i%2==0] #note: irange not range
print(x)
```

prints the squares of all the integers from 1 to 10 inclusive, but only if the integer is even (computed as leaving a remainder of 0 when divided by 2). Run it. Change it a bit to get is to do something else.





# Comprehensions

---

- Here is another example

```
x = [2*c for c in 'some text' if c in 'bcdfghjklmnpqrstvwxyz']  
print(x)
```

which prints a list with strings with doubled characters for all the consonants (no aeiouy -or spaces for that matter): ['ss', 'mm', 'tt', 'xx', 'tt'].

- We can translate any list comprehension into equivalent code that uses more familiar Python looping/if/list appending features.

```
x = []                                # start with an empty list  
for var in iterable:                  # iterate through iterable  
    if p(var):                         # if var is acceptable?  
        x.append(f(var))               # add f(var) next in the list
```



# Comprehensions

---

- But often using a comprehension (in the right places: where you want to generate some list, tuple, set or dict) is simpler. Not all lists that we build can be written as simple comprehensions, but the ones that can are often very simple to write, read, and understand when written as comprehensions.
- What comprehensions aren't good for is putting information into a data structure and then mutating/changing it during the execution of the comprehension; for that job you need code more like the for loop above. So when deciding whether or not to use a comprehension, ask yourself if you can specify each value in the data structure once, without changing it (as was done above, using comprehensions). So, try to write the code as a comprehension first; if you fail, then try to write it using more complicated statements in Python.



# Comprehensions

---

- Note that we can add-to (mutate) lists, sets, and dicts, but not tuples. For tuples we would have to write this code with `x = ()` at the top and `x = x + (var,)` in the middle: which builds an entirely new tuple by concatenating the old one and a one-tuple (containing only `x`) and then binding `x` to the newly constructed tuple. For large tuples, this process is very slow.
- Don't worry about these details, but understand that unlike lists, tuples have no mutator methods: so `x.append(...)` is not allowed.
- Here is something interesting (using a set comprehension: notice `{}` around the comprehension).

```
# note ' in str delimited by "  
x = {c for c in "I've got plenty of nothing"}  
print(sorted(x))
```



# Comprehensions

---

- It prints a set of all the characters (in a list, in sorted order, created by `sorted(x)`) in the string but because it is a set, each character occurs one time. So even though a few c's have the same value, only one of each appears in the set because of the semantics/meaning of sets. Note it prints

```
[' ', '"', 'I', 'e', 'f', 'g', 'h', 'i', 'l',  
'n', 'o', 'p', 't', 'v', 'y']
```

- If we used a list comprehension, the result would be much longer because, for example, the character 't' would occur 3 times in a list (but occurs only once in a set)



# Dict Comprehensions

---

- The form for dict comprehensions is similar, here `k` and `v` are functions (or expressions) using `var`. Notice the `{}` on the outside and the `:` on the inside, separating the key from the value. That is how Python knows the comprehension is a dict not a set.

```
{k(var, ...) : v(var, ...) for var in iterable if p(var, ...) }
```

So,

```
x = {k : len(k) for k in ['one', 'two', 'three', 'four', 'five']}
```

```
print(x)
```

prints a dictionary that stores keys that are these five words whose associated values are the lengths of these words. Because dicts aren't ordered, it could print as `{'four': 4, 'three': 5, 'one': 3, 'five': 4, 'two': 3}`



# Dict Comprehensions

---

- Finally, we can write a nested comprehension, although they are harder to understand than simple comprehensions.

```
x = {c for word in ['i', 'love', 'new', 'york'] \
      for c in word if c not in 'aeiou'}

print(x)
```

- It says to collect c's, by looking in each word w in the list, and looking at each character c in each word: so, the c collected at the beginning is the c being iterated over in the second part of the comprehension (for c in word...)
- It prints a set of each different letter that is not a vowel, in each word in the list. I could produce this same result by rewriting the outer part of the comprehension as a loop, but leaving the inner one as a comprehension (union merges two sets: does a bunch of adds).



# Dict Comprehensions

---

```
x = set()          # empty set: cannot use {} which is an empty dict
for word in ['i', 'love', 'new', 'york']:
    x = x.union( {c for c in word if c not in 'aeiou'} )
print(x)
```

or write it with no comprehensions at all

```
x = set()
for word in ['i', 'love', 'new', 'york']:
    for c in word:
        if c not in 'aeiou':
            x.add(c)
print(x)
```



# Dict Comprehensions

---

- So, which of these is the most comprehensible: the pure comprehension, the hybrid loop/comprehension, or the pure nested loops? What is important is that we know how all three work, can write each correctly in any of these ways, and then we can decide afterwards which way we want the code to appear. As we program more, our preferences might change. I'd probably prefer the first one (because I've seen lots of double comprehensions), but the middle one is also reasonable.
- What do you think the following nested (in a different way) comprehension produces?

```
x = {word : {c for c in word} for word in \
      ['i', 'love', 'new', 'york']}
```

- Check your answer by executing this code in Python and printing x.





# Comprehensions

---

- Finally, here is a version of myprint (written above in the section on the binding of arguments and parameters) that uses a combination of the .join function and a comprehension to create the string to print simply.
- The .join function (discussed more in depth below) joins all the string values in an iterable into a string using the prefix operator as a separator:

```
'--'.join( ['My', 'dog', 'has', 'fleas'] )  
returns the string 'My--Dog--has--fleas'
```

```
def myprint(*args, sep=' ', end='\n') :  
    s = sep.join(str(x) for x in args)+end #create string to print  
    print(s, end='')                      #print the string
```



# Comprehensions

---

**WARNING:** Once students learn about comprehensions, sometimes they go a bit overboard as they learn about/use this feature. Here are some warning signs:

When writing a comprehension, you should (1) use the result produced in a later part of the computation and (2) typically not mutate anything in the comprehension. If the purpose of your computation is to mutate something, don't use a comprehension. Over time you will develop good instincts for when to use comprehensions.



# Tuple Comprehensions are special

---

- The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over. We will discuss generators in detail later in the quarter, so for now we will examine just some simple examples. Given the code

```
x = (i for i in 'abc')    # tuple comprehension
print(x)
```

- You might expect this to print as ('a', 'b', 'c') but it prints as

```
<generator object <genexpr> at 0x02AAD710>
```



# Tuple Comprehensions are special

---

- The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE. So, given the code

```
x = (i for i in 'abc')
for i in x:
    print(i)
for i in x:
    print(i)
```

it prints

```
a
b
c
```



# Tuple Comprehensions are special

---

- Yes, it prints a, b, c and just prints it once: after the first loop finishes, the generator is exhausted so the second loop prints no more values. We will spend a whole studying the details of generators later in the quarter.
- Recall our discussion of changing any iterable into a list, tuple, set by iterating over it; we can iterate over a tuple comprehension. So if we wrote `t = tuple(x)` or `t = tuple( (i for i in 'abc') )`, then `print(t)` would print `('a', 'b', 'c')`. In fact, we could even write `t = tuple(i for i in 'abc')` because by default, comprehensions are tuple comprehensions.

- Of course, we could also write things like

```
l = list(i for i in 'abc')
s = set (i for i in 'abc')
```

but these are equivalent to writing the standard comprehensions more simply:

```
l = [i for i in 'abc']
s = {i for i in 'abc'}
```

# Nine Important/Useful Functions

LECTURE 2



# Nine Important/Useful Functions:

*split/join, any/all, sum/min/max, zip/enumerate*

---

## The split/join methods

- Both split and join are methods in the str class: if s is some string, we call them by writing `s.split(...)` or `s.join(...)`
- The split method also takes one str argument as .... and the result it returns is a list of str. For example `'ab;c;ef;;jk'.split(';')` returns the list of str  

```
['ab', 'c', 'ef', '', 'jk']
```
- It uses the argument str ';' to split up the string (prefixing the .split call), into slices that come before or after every ';'.



# Nine Important/Useful Functions:

**split/join, any/all, sum/min/max, zip/enumerate**

---

- Note that there is an empty string between 'ef' and 'j' because two adjacent semi-colons appear between them in the string. If we wanted to filter out such empty strings, we can easily do so by embedding the call to split inside a comprehension: writing `[s for s in 'ab;c;ef;;jk'.split(';') if s != '']` produces the list `['ab', 'c', 'ef', 'jk']`.
- Because the prefix and regular arguments are both strings, students sometime reverse these two operands: what does `(';').split('ab;c;ef;;jk')` produce and why?
- The split method is very useful to call after reading lines of text from a file, to parse (split) these lines into their important constituent information.
- You will use split in all 5 parts of Programming Assignment #1.





# Nine Important/Useful Functions:

**split/join, any/all, sum/min/max, zip/enumerate**

---

- The **join method** also takes one iterable (it must produce str values) argument as ....; the result it returns is a str. For example `' ; '.join(['ab', 'c', 'ef', '', 'jk'])` returns the str  
`'ab;c;ef;;jk'`
- It merges all the strings produced by iterating over its argument into one big string, with all the strings produced by iterating over its argument concatenated together and separated from each other by the ';' string.
- So, **split** and **join** are opposites. Unfortunately, the splitting/joining string ';' appears as the argument inside the () in split, but it appears as the prefix argument before the call in join. This inconsistency can be confusing.

# The all/any functions

LECTURE 3



# The all/any functions (and their use with tuple comprehensions)

---

- The **"all"** function takes one iterable argument (and returns a bool value): it returns **True** if ALL the bool values produced by the iterable are **True**; it can stop examining values and return a **False** result when the first **False** is produced (ultimately, if no False is produced it returns **True**).
- The **"any"** function takes one iterable argument (and returns a bool value): it returns **True** if ANY the bool values produced by the iterable are **True**; it can stop examining values and return a **True** result when the first **True** is produced (ultimately, if no **True** is produced it returns False).



# The all/any functions (and their use with tuple comprehensions)

---

- These functions can be used nicely with tuple comprehensions. For example, if we have a list `l` of numbers, and we want to know whether all these numbers are prime, we can call

```
all( predicate.is_prime(x) for x in l )
```

which is the same as calling

```
all( (predicate.is_prime(x) for x in l) )
```

and similar (but more time/space-efficient) than calling

```
all( [predicate.is_prime(x) for x in l] )
```



# The all/any functions (and their use with tuple comprehensions)

---

- The list comprehension computes the entire list of boolean values and then "all" iterates over this list. When "all" iterates over a tuple comprehension, the tuple comprehension computes values one at a time and "all" checks each: if one is False, the tuple comprehension returns False immediately and does not have to compute any further values in the tuple comprehension. The tuple comprehension version can be much more efficient, if a False value is followed by a huge number of other values.
- Likewise, for the "any" function, which produces True the first time it examines a True value.



# The all/any functions (and their use with tuple comprehensions)

---

- Here is how we can write these functions, which search for a False or a True respectively:

```
def all(iterable):
    for v in iterable:
        if v == False:
            return False    # something was False; return False immediately
    return True             # nothing was False; return True after loop
def any(iterable):
    for v in iterable:
        if v == True:
            return True     # something was True; return True immediately
    return False            # nothing was True; return False after loop
```

- Read these functions. What does each produce if the iterable produces no values?
- Why is that reasonable?

# The sum/max/min functions

LECTURE 4



# The sum/max/min functions

---

- The simple versions of the sum function takes one iterable argument. The sum function requires that the iterable produce numeric values that can be added. It returns the sum of all the values produced by the iterable; if the iterable argument produces no values, sum returns 0. Actually, we can supply a second argument to the sum function; in this case, that value will be returned when the iterable produces no values and if the iterable does produce values, the sum will be the actual sum plus this argument. We can think of sum as defined by

```
def sum(values, init_sum=0):  
    result = init_sum  
    for v in values:  
        result += v  
    return result
```





# The sum/max/min functions

---

- The simple versions of the max and min functions also each take one iterable argument.
- The min/max functions require that the iterable produce values that can be compared with each other; so calling `min([2,1,3])` returns 1; and calling `min(['b','a','c'])` returns 'a'; but calling `min([2,'a',3])` raises a `TypeError` exception because Python cannot compare an integer to a string. These functions return the minimum/maximum value produced by their iterable argument; if the iterable produces no values (e.g., `min([])`), min and max each raise a `ValueError` exception, because there is no minimum/maximum value that it can compute/return.
- There are two more interesting properties to learn about the min and max functions.



# The sum/max/min functions

---

- First, we can also call min/max specifying any number of arguments or if one argument, it must be iterable: so, calling `min([1,2,3,4])` -using a tuple which is iterable- produces the same result as calling `min(1,2,3,4)` -using 4 arguments.
- We can also specify a named argument in min/max: a key function just like the key function used in sort/sorted. The min/max functions return the smallest/largest value in its argument(s), but if the key function is supplied, it compares two values by calling the key function on each.
- For example, `min('abcd','xyz')` returns 'abcd', because 'abcd' < 'xyz'. But if we instead wrote `min('abcd','xyz',key = (lambda x : len(x)))` it would return 'xyz' because `len('xyz') < len('abcd')`: that is, it compares the key function applied to all values in the iterable to determine which value to return.



# The sum/max/min functions

---

- Note that `min('abcd','wxyz',key = (lambda x : len(x)) )` will return 'abcd' because it is the first value that has the smallest result when the key function is called: the key function returns 4 for both values. We can think of the min function operating on an iterable to be written as follows (although we will learn Python features later on that simplifies the actual definition of this function)



# The sum/max/min functions

---

```
def min(*args, key = (lambda x : x) ): # default key is the identity function
    if len(args) == 0:
        raise TypeError('min: expected >=1 arguments, got 0')
    if len(args) == 1:      # Assume that if min has just one argument
        args = args[0]      # it's iterable, so take the max over its values
    answer = None
    for v in args:
        key_of_v = key(v)
        if answer == None or key_of_v < key_of_answer:
            answer = v
            key_of_answer = key_of_v
    return answer
```



# The sum/max/min functions

---

- Calling `min( ('abc','def','gh','ij'),key = (lambda x : len(x)) )` returns 'gh'.
- Note that the default value for the key function is a lambda that returns the value it is passed: this is called the identity function/lambda.

# The zip and enumerate functions

LECTURE 5



iterable A



iterable B



zip(iterable A, iterable B)





# Zip

---

- There is a very interesting function called `zip` that takes an arbitrary number of iterable arguments and zips/interleaves them together (like a zipper does for the teeth on each side). Let's start by looking at just the two argument version of `zip`.
- What `zip` actually produces is a generator -the ability to get the results of zipping- not the result itself. See the discussion above about how tuple comprehensions produce generators.





# Zip

---

- So, to "get the result itself" we should use a for loop or constructor (as shown in most of the examples below) to iterate over the generator result of calling zip. The following code

```
z=zip('abc', (1, 2, 3)) # String and tuple are iterator arguments  
print('z:', z, 'list of z:', list(z))
```

prints

```
z: <zip object at 0x02A4D990> list of z: [('a', 1), ('b', 2),  
('c', 3)]
```



# Zip

---

- Here, z refers to a zip generator object; the result of using z in the list constructor is `[('a', 1), ('b', 2), ('c', 3)]` which zips/interleaves the values from the first iterable and the values from the second:

```
[ (first from first, first from second),  
  (second from first, second from second),  
  (third from first, third from second) ]
```

- What happens when the iterables are of different lengths? Try it.

```
z = zip( 'abc', (1, 2) )    # String and tuple for iterables  
print(list(z))             # prints [('a', 1), ('b', 2)]
```



# Zip

---

- So, when one iterable runs out of values to produce, the process stops. Here is a more complex example with three iterable parameters of all different sizes.

- Can you predict the result it prints: do so, and only then run the code.

```
z = zip( 'abcde', (1, 2, 3), ['1st', '2nd', '3rd', '4th'] )  
print(list(z))
```

which prints

```
[('a', 1, '1st'), ('b', 2, '2nd'), ('c', 3, '3rd')]
```

- Of course, this generalizes for any number of arguments, interleaving them all (from first to last) until any iterable runs out. So, the number of values in the result is the minimum of the number of values of the argument iterables.



# Zip

---

- **Note** one very useful way to use zip: suppose we want to iterate over values in two iterables simultaneously, i1 and i2, operating on the first pair of values in each, the second pair of values in each, etc. We can use zip to do this by writing:

```
for v1,v2 in zip(i1,i2):  
    process v1 and v2: the next pair of values in each
```

- So

```
for v1,v2 in zip ( ('a','b','c'), (1,2,3) ):  
    print(v1,v2j)
```

prints

```
a 1  
b 2  
c 3
```



# Zip

---

- Using zip, we can write a small function that computes the equivalent of the < (less than) operator for strings in Python (see the discussion above about the meaning of < for strings).

```
def less_than(s1 : str, s2 : str)-> bool:
    for c1,c2 in zip(s1,s2):    # examine 1st, 2nd, ... characters of each string
        if c1 != c2:           # if current characters are different
            return c1 < c2      # compute result by comparing characters
    # if all character from the shorter are the same (as in 'a' and 'ab')
    # return a result based on the length of the strings
    return len(s1)<len(s2)
```



finxter

# enumerate(iterable, start=0)

Generate (counter, element) pairs for each element in *iterable*, starting to count from *start* (per default *start=0*).

```
fruits = ['apple', 'banana', 'cherry']

for i in range(len(fruits)):
    print(i, fruits[i])

# Output:
# 0 apple
# 1 banana
# 2 cherry
```



*Not Pythonic*

```
fruits = ['apple', 'banana', 'cherry']

for i, fruit in enumerate(fruits):
    print(i, fruit)

# Output:
# 0 apple
# 1 banana
# 2 cherry
```



*Pythonic*



# Enumerate

---

- Finally, this is a convenient time to toss in another important function: `enumerate`. It also produces a generator as a result, but has just one iterable argument, and an optional 2nd argument that is a starting number. It produces tuples whose first values are numbers (starting at the value of the second parameter; omit a 2nd parameter and unsurprisingly, the starting number is 0) and whose second values are the values in the iterable. So, if we write

```
e = enumerate(['a', 'b', 'c', 'd'], 5)
print(list(e))
```

it prints `[(5, 'a'), (6, 'b'), (7, 'c'), (8, 'd')]`



# Enumerate

---

- Given `l = ['a','b','c','d','e']` we could write the following code

```
for i in range(len(l)) :  
    print(i+1,l[i])
```

(which prints 1 a, 2 b, 3 c, 4 d, and 5 e on separate lines) more simply by using `enumerate` (notice the use of parallel/tuple assignment for `i,x`):

```
for i,x in enumerate(l,1) :  
    print(i,x)
```





# Enumerate

---

- Another nice example illustrating enumerate is reading a file a line at a time, and processing the line number and the line contents.

```
line_number = 1
for line in open("file-name"):
    .... process line_number and line
    line_number += 1
```

- We can write just

```
for line_number, line in enumerate(open("file-name"), 1):
    .... process line_number and line
```



# Enumerate

---

- You might ask now, why do these things (tuple comprehension, zip, enumerate) produce generators and not just tuples or lists? That is an excellent question.
- It goes right along with the excellent question why does `sorted(...)` produce a list and not an iterable? We will discuss these issues later in the quarter.
- The generator question mostly has to do with space efficiency when iterating over very many values. The sorted question has to do with why there is no way to do this operation in a space efficient way.

`**kwargs` for dictionary

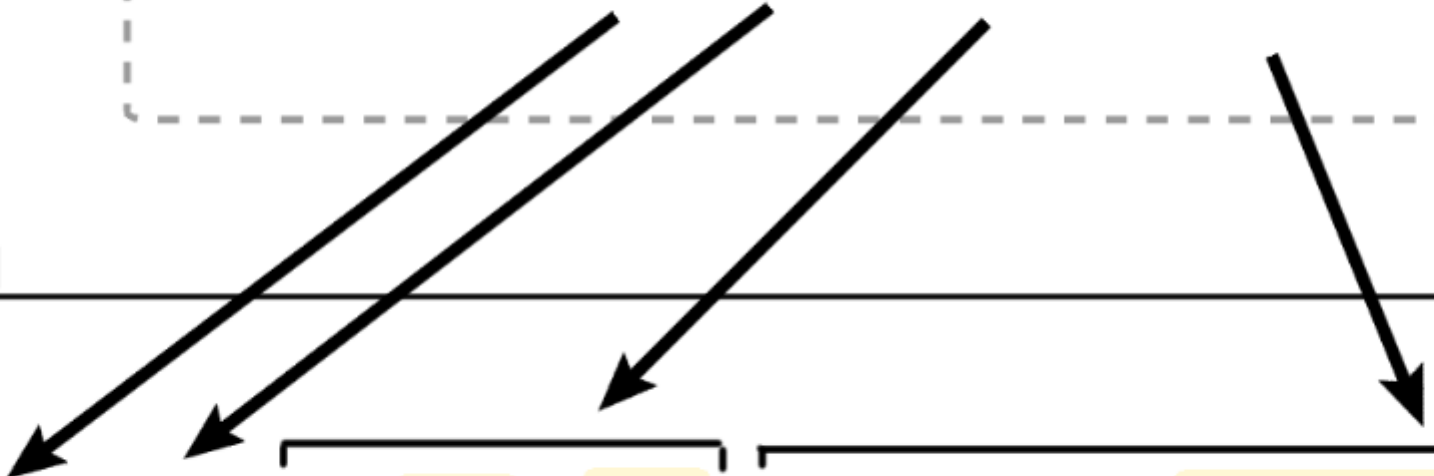
LECTURE 6

## Definition

```
def function( arg1, arg2, *args, **kwargs ):
```

## Function Call

```
function( val1, val2, val3, val4...valN, name1 = valx, name2= valy...nameN=valz )
```





# `**kwargs` for dictionary

---

- Recall the use of `*args` in a function parameter list. We can also write the symbol `**kwargs` (we can write `**` and any word, but `kwargs` or `kwargs` are the standard ones). If we use it to specify this kind of parameter, it must occur as the last parameter. `kwargs` stands for keyword arguments.
- Basically, if Python has any keywords arguments that do not match keyword parameters (see the large discussion of argument/parameter binding above, which includes `*args` but doesn't include `**kwargs`) they are all put in a dictionary that is stored in the last parameter named `kwargs`.



# `**kargs` for dictionary

---

- So, imagine we define the following function

```
def f(a,b, **kargs):  
    print(a,b,kargs)
```

and call it by

```
f(c=3, a=1, b=2, d=4)
```

it prints: 1 2 {'c': 3, 'd': 4}

- Using the rules specified before, Python would report a **TypeError** exception (by rule M5(d)), because there are no parameter named c or d.



# `**kargs` for dictionary

---

- By the new rules, Python finds two named arguments (`c=3` and `d=4`) whose names did not appear as parameter names in the function header of `f` (which specifies only `a` and `b`, and of course the special `**kargs`), so while Python directly binds `a` to `1` and `b` to `2` (the parameter names specified in the function header, matched to similarly named arguments) it creates a dictionary with all the other named arguments: `{'c': 3, 'd': 4}` and binds `kargs` to that dict.

- The same result would be printed for the call

```
f(1, 2, d=4, c=3)
```

- We will use `**kargs` to understand a special use of dict constructors (below).
- We will also use `**kargs` (and learn something new in the process) when discussing how to (1) call methods in decorators and (2) call overridden methods using inheritance much later in the quarter.



# `**kargs` for dictionary

---

- Note that to write a perfectly general function that is called with any kinds of arguments we can write

```
def g(*args, **kargs):  
    print(args, kargs)
```

- Now, any legal call of `g` (one with any legal combination of positional and named arguments) will populate the `*args` and `**kargs` structures appropriately.

- Calling

```
g(1, 2, c=3, d=4)      prints (1, 2) {'c': 3, 'd': 4}
```

```
g(a=1, b=2, c=3, d=4) prints () {'c': 3, 'b': 2, 'a': 1, 'd': 4}
```





# `**kargs` for dictionary

---

- Generally, when a parameter name is prefixed by `*` and `**` in a function header, we omit the prefix when we use the parameter name inside the function. But there are times where we use the `*` and `**` prefixes.

```
def h(a,b,c,d):  
    print(a,b,c,d)  
def i(*args,**kargs):  
    h(*args,**kargs)
```

- The arguments `*args` and `**kargs` in the call of `h` expand the `args` tuple to be positional arguments and the `**kargs` dictionary to be named arguments



# `**kargs` for dictionary

---

Calling

`i(1,2,c=3,d=4)` prints 1 2 3 4: it calls `h(1,2,c=3,d=4)`

`i(a=1,b=2,c=3,d=4)` prints 1 2 3 4: it calls `h(c=3,b=2,a=1,d=4)`



# Summarizing:

---

- Writing `*` and `**` when specifying parameters makes those parameters names bind to a tuple/dict respectively.
- Using the parameter names by themselves in the function is equivalent to using the tuple/dict respectively.
- Using `*` and `**` followed by the parameter name as arguments in function calls expands all the values in the tuple/dict respectively to represent all the arguments.

Experiment with `*args/**kargs` as parameters of functions and `args/kargs` and `*args/**kargs` as arguments to other function calls.

# Lists, Tuples, Sets, Dictionaries

LECTURE 7



# Lists, Tuples, Sets, Dictionaries

---

- You need to have pretty good grasp of these important data types, meaning how to construct them and the common methods/operations we can call on them. Really you should get familiar with reading the online documentation for all these data types (see the Python Library Reference link on the homepage for the course).

4. 6: Sequence Types includes Lists (mutable) and Tuples (immutable)

4. 9: Set Types includes set (mutable) and frozenset (immutable)

4.10: Mapping Types includes dict and defaultdict (both mutable)

Here is a very short/condensed summary of these operations. Experiment with them in Eclipse.



# Sequence Types includes Lists (mutable) and Tuples (immutable)

These sequence operations (operators and functions) are defined in 4.6.1

```
x in s, x not in s, s + t, s * n, s[i], s[i:j], s[i:j:k], len(s), min(s),  
max(s), s.index(x[, i[, j]]), s.count(x)
```

Mutable sequence allow the following operations, defined in 4.6.3

```
s[i] = x , s[i:j] = t, del s[i], s[i:j:k] = t, del s[i:j:k], s.append(x)  
s.clear(), s.copy(), s.extend(t), s.insert(i, x), s.pop(), s.pop(i),  
s.remove(x), s.reverse()
```

It also discusses list/tuple constructors and sort for list.

note that the append method is especially important for building up sequences like lists. Also to return and remove a value from a sequence, we call

```
x = s.pop(i)
```

is equivalent to

```
x = s[i]  
del s[i]
```

and calling s.pop() uses the value len(s)-1 (the last index in the sequence)



# Set Types includes set (mutable) and frozenset (immutable)

---

These set (operators and functions) are defined in 4.6.1.9 `len(s)`, `x in s`, `x not in s`, `isdisjoint(other)`, `issubset(other)`, `set <= other`, `set < other`, `issuperset(other)`, `set >= other`, `set > other`, `union(other, ...)`, `intersection(other, ...)`, `difference(other, ...)`, `symmetric_difference(other)`, `copy`; also the operators `|` (for union), `&` (for intersection), `-` (for difference), and `^` (for symmetric difference)

Sets, which are mutable, allow the following operations `update(other, ...)`, `intersection_update(other, ...)`, `difference_update(other, ...)`, `symmetric_difference_update(other)`, `add(elem)`, `remove(elem)`, `discard(elem)`, `pop()`, `clear()`; also the operators `|=` (union update), `&=` (intersection update), `-=` (difference update), `^=` (symmetric difference update)



# Mapping Types includes dict and defaultdict (both mutable)

---

These dict (operators and functions) are defined in 4.10

`d[key] = value` , `del d[key]`, `key in d`, `key not in d`, `iter(d)`, `clear()`, `copy()`,  
`fromkeys(seq[, value])`, `get(key[, default])`, `items()`, `keys()`,  
`pop(key[, default])`, `popitem()`, `setdefault(key[, default])`, `update([other])`,  
`values()`



```
from collections import defaultdict
```

```
int_dct = defaultdict(int) → defaultdict of int  
int_dct['Key - 1']
```

```
list_dct = defaultdict(list) → defaultdict of list  
list_dct['Key - 1']
```

```
print (int_dct) → {'Key - 1': 0}  
print (list_dct) → {'Key - 1': []}
```

```
int_dct['Key - 1'] = "Hello"  
list_dct['Key - 1'] = "World"
```

```
print (int_dct) → {'Key - 1': 'Hello'}  
print (list_dct) → {'Key - 1': 'World'}
```



# Important Notes on dicts:

---

**d[k]** returns the value associated with a key (raises exception if k not in d)

**d.get(k,default)** returns d[k] if k in d; returns default if k not in d

it is equivalent to the conditional expression (d[k] if k in d else default)

**d.setdefault(k,default)** returns d[k] if k in d; if k not in d it

(a) sets d[k] = default

(b) returns d[k]

writing **d.setdefault(k,default)** is equivalent to (but more efficient than) writing

```
if k in d:
    return d[k]
else:
    d[k] = default
    return d[k]
```



# defaultdict

---

- There is a type called defaultdict (see 8.3.4) whose constructor generally takes an argument that is a reference to any object that CAN BE CALLED WITH NO ARGUMENTS. Very frequently we use a NAME OF A CLASS that when called will CONSTRUCT A NEW VALUE: if the argument is int, it will call int() producing the value 0; if the argument is list, it will call list() producing an empty list; if the argument is set, it will call set() producing an empty set; etc.
- Whenever a key is accessed for the first time (i.e., that key is accessed but not already associated with a value in the dictionary) in a defaultdict, it will associate that key with the value created by calling the reference to the object supplied to the constructor.



# defaultdict

---

Here is an example of program first written with a dict, and simplified later by using a defaultdict.

```
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict()          # could use = {}
for l in letters:
    # must check l in freq_dict before freq_dict[l]
    if l not in freq_dict:
        freq_dict[l] = 1    # if not there, put with frequency of 1
    else:
        freq_dict[l] += 1   # otherwise there, increment frequency
print(freq_dict)
```

This would print the following dict: {'b': 1, 'x': 3, 'f': 1, 'a': 2}



# defaultdict

---

- As each letter in the loop is processed, it is associated with 1 (if not already in the dict) or it is in the dict, and its associated value is incremented by 1.
- We could solve this a bit more easily with a defaultdict.

```
from collections import defaultdict # in same module as namedtuple
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = defaultdict(int)
# int not int(); but int() returns 0 when called
for l in letters:
    freq_dict[l]+=1 # in dict, exception raised if l not in d, but
print(freq_dict)   # defaultdict calls int() putting 0 there first
```



# defaultdict

---

- As each letter in the loop is processed, its associated key is looked up: if the key is absent, it is placed in the dict associated with `int()/0`; then the associated value (possibly the one just put in the dict) is incremented by 1.
- The dict code below is equivalent to how the defaultdict code above works.

```
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict() # could use = {}
for l in letters:
    if l not in freq_dict:
        # must check l in freq_dict before freq_dict[l]
        freq_dict[l] = int()
        # int() constructor returns 0; could write 0 here
    freq_dict[l] += 1
    # l is guaranteed in freq_dict, either because
print(freq_dict)
# it was there originally, or just put there
```



# defaultdict

---

- Another way to do the same thing (but also a bit longer and less efficient) uses the setdefault method (listed above)

```
letters = ['a', 'x', 'b', 'x', 'f', 'a', 'x']
freq_dict = dict()          # note dict only
for l in letters:
    freq_dict[l] = freq_dict.setdefault(l,0) + 1
print(freq_dict)
```



# defaultdict

---

- Here we evaluate the right side of the = first; if `l` is already in the dict, its associated value is returned; if not, `l` is put in the map with an associated value of 0, then this associated value is returned (and then incremented, and stored back into `freq_dict[l]` replacing the 0 just put there).
- You should achieve a good understanding of why each of these four scripts work and why they all produce equivalent results.
- Often, we use defaultdicts with list instead of int: just as `int()` produces the object 0 associated with a new key, `list()` creates an empty list associated with a new key (and later we add things to that list); likewise, we can use defaultdicts with set to get an empty set with a new key.





# defaultdict

---

- So, if we wrote `x = defaultdict(list)` and then immediately wrote `x['bob'].append(1)` then `x['bob']` is associated with the list `[1]` (the empty list, with 1 appended to it). If we then wrote `x['bob'].append(2)`, then `x['bob']` is associated with the list `[1, 2]`. So, we can always append to the value associated with a key, because it will use an empty list to start the process if there is nothing associated with a key.
- Just as with comprehensions, what is important is that we know how things like defaultdicts and dicts (with the `setdefault` method) work, so that we can correctly write code in any of these ways. We can decide afterwards which way we want the code to appear. As we program more, our preferences might change. I have found defaultdicts are mostly what I use to simplify my code, but every so often I must use a regular dict.
- Later in the quarter we will use inheritance to show how to write the defaultdict class simply, by extending the dict class.

# Printing Dictionaries in Order

LECTURE 8



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- In this section we will combine our knowledge about the sorted function, comprehensions, and iterating over dictionaries to examine how we can print (or generally process) dictionaries in arbitrary orders.

- Generally, all of our code will be of the form

```
for index(es) in sorted( iterable, key = (lambda x : ....) ):
    print( index(es) )
```

- In these examples, we will use the simple dictionary

```
d = { 'x': 3, 'b': 1, 'a': 2, 'f': 1 }
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- In each example, we will discuss the relationships among index(es), iterable, and the (lambda x : ....) in the function bound to the key parameter.

1. In the first example, we will print the dictionary keys in increasing alphabetical order, and their associated values, by iterating over d.items().

```
for k,v in sorted( d.items(), key = (lambda item : item[0]) ):
    print(k, '->', v)
```

which prints as

```
a -> 2
b -> 1
f -> 1
x -> 3
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- Here, iterable is `d.items()`, which produces 2-tuples storing each key and its associated value, for every item in the dictionary; the item in `lambda item` is also a key/value 2-tuple (specifying here to sort by `item[0]`, the key part in the 2-tuple); finally, the list returned by `sorted` also contains key/value 2-tuples, which are unpacked into `k` and `v` and printed.

- We can solve this same problem by iterating over just the keys in `d` as well.

```
for k in sorted( d.keys(), key = (lambda k : k) ):
    print(k, '->', d[k])
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- Here, iterable is `d.keys()` which produces strings storing each key in the dictionary; the `k` in `lambda k` is also a key/str value (specifying here to sort by `k`, the key itself: I could have omitted this identity `lambda`); finally, the list returned by `sorted` also contains key/str value, which are stored into `k` and printed along with `d[k]`.
- This code is equivalent to the following, since `d.keys()` is the same as just `d`, and `lambda x : x` is the default for the `key` parameter.

```
for k in sorted( d ) :  
    print(k, '->', d[k])
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

2. In the second example, we will print the dictionary keys and their associated values, in increasing order of the values, by iterating over `d.items()`.

```
for k,v in sorted( d.items(), key = (lambda item :  
item[1]) ) :  
    print(k, '->', v)
```

which prints as

```
b -> 1  
f -> 1  
a -> 2  
x -> 3
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- Here, iterable is `d.items()`, which produces 2-tuples storing each key and its associated value, for every item in the dictionary; the item in lambda item is also a key/value 2-tuple (specifying here to sort by `item[1]`, the value part in the 2-tuple); also, the list returned by `sorted` also contains key/value 2-tuples, which are unpacked into `k` and `v` and printed. Finally, by this lambda either `b` or `f` might be printed first, because they have the same value associated with them.

- We can solve this same problem by iterating over just the keys in `d` as well.

```
for k in sorted( d.keys(), key = (lambda k : d[k]) ):
    print(k, '->', d[k])
```





# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

- Here, iterable is `d.keys()` which produces strings storing each key in the dictionary; the `k` in `lambda k` is also a key/str value (specifying here to sort by `d[k]`, the value associated with the key); finally, the list returned by `sorted` also contains key/str values, which are stored into `k` and printed along with `d[k]`.
- This code is equivalent to the following, since `d.keys()` is the same as `d`; the `lambda` is still needed here because it is not the identity `lambda`.

```
for k in sorted( d, key = (lambda k : d[k]) ) :  
    print(k, '->', d[k])
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

3. In the third example, we will compute a list that contains all the keys in a dictionary, sorted by their associated values (in decreasing order of the values), by iterating over `d.keys()` (really just `d`, to simplify the code)

- We compute

```
ks = sorted(d, key = (lambda k : d[k]), reverse = True)
ks stores ['x', 'a', 'b', 'f'].
```

- We could also compute this list less elegantly using `d.items()`.; I say less elegantly, because we need a comprehension to "throw away" the value part of each item returned by `sorted`.

```
ks = [k for k,v in sorted(d.items(),
                        key=(lambda item : item[1]), reverse=True)]
```



# Printing Dictionaries in Order:

## An example of using comprehensions and sorted

---

4. Finally, in the fourth example, we will compute a list that contains all the 2-tuples in the items of d, but the tuples are reversed (values before keys) and sorted in increasing alphabetical order by keys.

- We compute this list of 2-tuple in the following two ways

```
vks = [ (v,k) for k,v in sorted( d.items(), key = (lambda item : item[0]) ) ]  
vks = sorted( ((v,k) for k,v in d.items()), key = (lambda item : item[1]) )  
vks stores [(2, 'a'), (1, 'b'), (1, 'f'), (3, 'x')]
```

- The top computation first creates a sorted version of d by keys, and then uses a comprehension to create tuples that reverse the keys/values; the bottom computation first uses a comprehension to create a list of reversed keys/values, and then uses sorted to sort these reversed 2-tuples by the keys that are now in the second part of each tuple.

# Exceptions

LECTURE 9



# Exceptions:

## example from `prompt_for_int`

---

- We do two things with exceptions in Python: we raise them (with the `raise` statement) and we handle them (with the `try/except` statement). Exceptions were not in early programming languages, but were introduced big time in the Ada language in the early 1980s, and have been parts of most new languages since then.
- A function raises an exception if it cannot do the job it is being asked to do. Rather than fail silently, possibly producing a bogus answer that gets used to compute a bogus result, it is better that the computation announces a problem occurred and if no way is known to recover from it (see handling exceptions below) the computation halts with an error message to the user.



# Exceptions:

## example from prompt\_for\_int

---

- For example, if your program has a list `l` and you write `l[5]` but `l` has nothing stored at index 5 (its length is smaller), Python raises the `IndexError` exception. If you haven't planned for this possibility, and told Python how to handle the exception and continue the calculation, then the program just terminates and Python prints:

```
IndexError: list index out of range
```

- Why doesn't it print the index value 5 and the length of list `l`? I don't know. That certainly seems like important and useful information, and Python knows those two values. I try to make my exception messages include any information useful to the programmer.



# Exceptions:

## example from prompt\_for\_int

---

- There are lots of ways to handle exceptions. Here is a drastically simplified example from my prompt class (this isn't the real code, but a simplification for looking at a good use of exception handling). But it does a good job of introducing the try/except control form which tries to execute a block of code and handles any exceptions it raises.
- If we write a try/except and specify the name of no exception, it will handle any exception. The name `Exception` is the name of the most generic exception.
- We can write a try/except statement with many excepts, each one specifying a specific exception to handle, and what to do when that exception is raised. In fact, `int('x')` raises the **`ValueError`** exception, so I use **`ValueError`** in the except clause below to be specific, and not accidentally handle any other kind of exception.



# Exceptions:

## example from prompt\_for\_int

---

```
def prompt_for_int(prompt_text):  
    while True:  
        try:  
            response = input(prompt_text+' : ')  
            # response is used in except  
            answer = int(response)  
            return answer  
        except ValueError:  
            print('  You bozo, you entered "', response, '"', sep='')  
            print('  That is not a legal int')  
print(prompt_for_int('Enter a positive number'))
```





# Exceptions:

## example from `prompt_for_int`

---

- So, this is an "infinite" while loop, but there is a return statement at the bottom of the try-block; if it gets executed, it returns from the function, thus terminating the loop. The loop body is a try/except; the body of the

```
try/except
```

1. prompts the user to enter a string on the console (this cannot fail)
2. calls `int(response)` on the user's input (which can raise the `ValueError` exception, if the user types characters that cannot be interpreted as an integer)
3. if that exception is not raised, the return statement returns an `int` object representing the integer the user typed as a string



# Exceptions:

## example from prompt\_for\_int

---

- But if the exception is raised, it is handled by the except clause, which prints some information. Now the try/except is finished, but it is in an infinite loop, so it goes around the loop again, reprompting the user (over and over until the user enters a legal int).
- Actually, the body of try could be simplified (with the same behavior) to just

```
response = input(prompt_text+' : ')\n# response is used in except\nreturn int(response)
```



# Exceptions:

## example from prompt\_for\_int

---

- If an exception is raised while the return statement is evaluating the int function, it still gets handled in except. We CANNOT write it in one line because the name response is used in the except clause (in the first print).

- If this WASN'T the case, we could write just

```
return int(input(prompt_text+' : '))
```

```
#if response not used in except
```

- For example, we might just say 'Illegal input' in the except, but in the example above, it actually display the string (not a legal int) that the user typed.
- Finally, in Java we throw and catch exceptions (obvious opposites, instead of raise and handle) so I might sometimes use the wrong term. That is because I think more generally about programming than "in a language", and translate what I'm thinking to the terminology of the language I am using, but, sometime, I get it wrong.

# Name Spaces

LECTURE 10



# Name Spaces (for objects): `__dict__`

---

- Every object has a special variable named `__dict__` that stores all its namespace bindings in a dictionary. During this quarter we will systematically study class names that start and end with two underscores. Writing `x.a = 1` is similar to writing `x.__dict__['a'] = 1`; both associate a name with a value in the object. We will explore the uses of this kind of knowledge in much more depth later in the quarter.
- Here is a brief illustration of the point above. Note that there is a small Python script that illustrates the point. This is often the case.



# Name Spaces (for objects): `__dict__`

---

```
class C:
    def __init__(self): pass

o = C()
o.a = 1
print(o.a)                # prints 1

o.__dict__['a'] = 2
print(o.a)                # prints 2

o.__dict__['b'] = 3
print(o.a, o.b)           # prints 2 3
```



# Trivial Things.

---

- An empty dict is created by `{}` and empty set by `set()` (we can't use `{}` for an empty set because Python would think it is a dict). Any non-empty dicts can be distinguished from a non-empty set because a non-empty dict will always have the `:` character inside `{}` separating keys from values. Suppose you want to create a set containing the value 1: which works (and why)? `{1}` or `set(1)`.
- A one value tuple must be written like `(1,)` with that "funny" comma (we can't write just `(1)` because that is just the value 1, not a tuple storing just 1).