Python Intermediate Programming

Unit 2:Basic Python Algorithms

CHAPTER 5: FUNCTIONAL PROGRAMMING AND COMBINATIONAL COMPUTING

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Functional programming is a style of programming that uses the simplicity and power of functions to accomplish programming tasks. Some talk about the functional programming paradigm. They contrast it with the more standard imperative paradigm (which includes both the procedural and object-oriented styles). Functional programs fundamentally evaluate expressions to compute results; imperative programs fundamentally execute statements to compute results (which often mutate data structures and rebind the values of names - which pure functional programming prohibits).

Objectives

- In a purely functional solution to a problem, there will be no mutation to data structures (instead of mutating one, a new one is built/returned, with the required changes: just as we have seen with strings, which are immutable), and recursion (not looping) is the primary control structure. Functions are called "referentially transparent": given the same inputs, they always produce the same result; we can always replace a function call with its ultimately returned result. Whether or not a computation occurs doesn't affect later computations (it has no side-effects, just a returned result).
- A certain class of functions, called tail-recursive (we will illustrate them in more detail below), can be automatically translated into non-recursive functions that run a bit faster and do not use extra space for call frames: each recursive function call occupies space for its parameters; most purely functional languages implement tail-recursive functions by doing this translation; Python does not (at least not yet).

Objectives

- Functional languages easily support and frequently use the passing of functions as arguments to other functions (these other functions are called "higher-order“ functions or functionals) and functions being returned from other functions as their values; we have seen both of these features used in Python, which does support these features well, and we will see more uses of these features in this lecture.
- There are programming languages that are purely function (Haskell), others that are mostly functional (ML -whose major implementations are SML and OCaml- the Scheme dialect of Lisp, and Erlang), and still others that at least support a functional style of programming (some better, some worse) when it is useful.
- Python is in this latter category, although features like comprehensions in in Python emphasize its functional programming aspects (generators and even lambdas fall into this category too).

Objectives

- Generally, functional programming is characterized as using immutable objects and no state changes (we can bind names only once, and not rebind them). Strings, tuples and frozensets are all immutable objects in Python (which means we can use their values as keys in dicts and values in sets), but lists, sets, and dicts are mutable (and therefore cannot be used these ways in sets and dicts).
- In functional programming, we don't change data structures but produce new ones that are variants of the old ones. For example, if we want to "change" the first value (at index 0) of a tupel t to 1, we instead create a new tuple whose first values is 1 and whose other values are all the other values originally in the tuple, using the concatenation operator. The new tuple is constructed as (1,)+t[1:]; note that we need the comma in (1,) to distinguish it from (1): the former is a tuple containing the value 1, the later is just a parenthesized int.

Objectives

- Functional programming creates lots of objects and must do so in a time and space efficient way, and for the most part, functional languages achieve parity in time/space efficiency with imperative programming languages (functional programs can be a bit slower, but they are often clearer/simpler/easier to both understand and modify). Mixed paradigm languages like Python tend not to do as well when used functionally as true functional languages. Emerging multi-paradigm languages like Scala and Clojure are closing the gap. Also, because of the simplicity of the semantics of functional programming, it is easier to automatically transform functional programs to run more efficiently in parallel, on cluster or multi-core computers.

Objectives

- Most functional programming languages are also statically type-safe. Before programs are executed they are compiled (type-checked). If they type-check correctly, the system executing them can be guaranteed that all operators and functions will be applied to the correct number and type of arguments; if not the compiler will report where errors like these are detected -the same errors that Python would find when it runs the code, but static-type checking occurs before any code runs. Contrast this with Python, which often discovers inappropriate argument types while running programs (at runtime), not before running them. Of course, there can still be other kinds of errors (not type-related) in functional programs: e.g., using a + operator where a * is needed.

Objectives

- Functional programming languages are still not as widely used as imperative languages, but they continue to find many uses in industry, and in some industries (telecommunications) they have achieved dominance (at least with some companies within the industries). Programmers who are trained to use functional languages think about problems and solve problems differently. All CS students should be exposed to functional programming as part of their education (and I mean an exposure longer than one day, or even a few weeks).
- To learn more about Python's use of functional programming, read section 10 (Functional Programming Modules) in Python's Library documentation, discussing the itertools, functools, and the operator modules. Some of this material is discussed in more detail below.

Map/Transform, Filter, Reduce/Accumulate

LECTURE 1

Map/Transform, Filter, Reduce/Accumulate:

- We start this lecture by looking at just three important higher-order functions used in functional programming: map (aka transform), filter, and reduce (aka accumulate). Each operates on a function and an iterable, which means they operate on lists and tuples easily, but also on iterables that don't store all their values and just produce values as necessary (e.g., the ints and primes generators).
- We will write recursive and generator versions of each, with the recursive versions having list parameters and returning lists, because many functional programing languages use only lists, not tuples, but lists are immutable in these languages: a base case for both lists and tuple is `len(x) == 0:` instead of `x==[]` or `x==()`.

Map/Transform, Filter, Reduce/Accumulate:

1) map/transform: this function takes a unary function and some list/iterable of values and produces a list/iterable of mapped/transformed values based on substituting each value with the result of calling the parameter function on that value. For example, calling

```
map_l(lambda x : x**2, [i for i in irange(0,5)])
```

takes a list as a second argument and produces a list as a result: a list of the squares of the values of the numbers 0 to 5: [0,1,4,9,16,25]. Calling

```
map_i(lambda x : x**2, irange(0,5))
```

takes a more general iterable as a second argument and produces an iterable as a result: an iterable of the squares of the values of the numbers 0 to 5.

Map/Transform, Filter, Reduce/Accumulate:

- So, the value produced is a generator that we can iterate over. If we wrote list(map_i(lambda x : x**2, irange(0,5))) Python would return the same result as for map_l because it would construct a list by iterating over the iterable thta map_i returned. But if we wrote

```
for i in map_i(lambda x : x**2, irange(0,5)):
    do something with i
```

- No list would be produced when executing the for loop (just as no list would be produced with executing: for i in range(0,5): ...

Map/Transform, Filter, Reduce/Accumulate:

- Note that lambdas are frequently (but not exclusively) used in calls to the map function: often we need to use a small, simple function once, which we can most easily write as a lambda. But, we can pass it any object that is callable, including function objects and class objects that implement __call__.
- Here are simple implementations of the list/iterator versions of this map. For map_l we define a recursive verion; map_i is a generate

```
def map_l(f,alist):
    if alist == []:
        return []
    else:
        return [f(alist[0])] + map_l(f,alist[1:])
```

Map/Transform, Filter, Reduce/Accumulate:

Note: no local variables and no mutation: + build a new list from existing ones. In languages that do not have comprehensions, this is the standard definition. In Python, which has comprehensions, we could also write it as

```
def map_l(f,alist):
    return [f(i) for i in alist]
```

- Here is the map_i function, which is a generator.

```
def map_i(f,iterable):
    for i in iterable:
        yield f(i)
```

Map/Transform, Filter, Reduce/Accumulate:

- Note that Python actually defines its map implementation to be a generator (so it is closer to map_i than map_l):

```
y = map(lambda x : x**2, irange(0,5))
print(y)
```

prints

```
<map object at 0x02C42BF0>
```

- This result is similar to printing the result of calling a a generator. It returns an object that we can iterate over.

Map/Transform, Filter, Reduce/Accumulate:

- Again, we can use a list/tuple/set constructor to turn such a map object into an actual list/tuple/set: it iterates through the values produced by map and collects these values in a list/tuple/set.

```
print(list(y))
```

prints

```
[0, 1, 4, 9, 16, 25]
```

Map/Transform, Filter, Reduce/Accumulate:

- In fact, the real map defined in Python is generalized to work on any number of lists/iterables. If there are n iterables, then the function f must have n parameters. So, if we called the real map function in Python (which as we've seen, produces an iterable) as

```
print(list( map(lambda x,y: x*y, 'abcde', irange(1,5))))
```

prints

```
['a', 'bb', 'ccc', 'dddd', 'eeeee']
```

- How does Python define map for these arbitrary number of arguments? It uses zip, which is actually a generator itself (returning something that is iterable). So, we can define Python's map function as a version of map_i generalized to arguments with multiple iterables.

Map/Transform, Filter, Reduce/Accumulate:

```
def map(f,*iterables):
    for args in zip(*iterables):
        yield f(*args)
```

- Recall that writing *iterables/*args INSIDE the body of a function/generator calls separates all the tuple components into positional arguments; writing *iterables in map's header combines any number of positional arguments into a tuple

again,

```
print( map(lambda x,y: x*y, 'abcde', irange(1,5)))
```

prints like <map object at 0x02C36EF0>.

Map/Transform, Filter, Reduce/Accumulate:

(2) filter: this function takes a predicate (a unary function returning a bool, although in Python most values have a bool interpretation: recall the __bool__ method) and some list/iterable of values and produces a list/iterable with only those values for which the predicate returns True (or a value that is interpreted as True by the __bool__ method defined in its class). For example, calling

```
import predicate
filter_l(predicate.is_prime, [i for i in irange(2,50)])
```

produces a list of the values between 2 and 50 inclusive that are prime:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47].
```

Map/Transform, Filter, Reduce/Accumulate:

- Here are simple implementations of the list/iterator versions of this function.
- Again, for filter_l we define a recursive verion; filter_i is a generator.

```
def filter_l(p,alist):
    if alist == []:
        return []
    else:
        if p(alist[0]):
            return [alist[0]] + filter_l(p,alist[1:])
        else:
            return filter_l(p,alist[1:])
```

Map/Transform, Filter, Reduce/Accumulate:

- Note: no local variables and no mutation: + build a new list from existing ones.
- We can simplify a bit, by using a conditional expression and noting that

```
[] + alist = alist
```

```
def filter_l(p,alist):
    if alist == []:
        return []
    else:
        return ([alist[0]] if p(alist[0]) else [])+    \
                filter_l(p,alist[1:])
```

Map/Transform, Filter, Reduce/Accumulate:

- In languages that do not have comprehensions, this is the standard definition.
- In Python, which has comprehensions, we could also write it as

```
def filter_l(p,alist):
    return [i for i in alist if p(i)]
```

- Here is the filter_i function, which is a genrator.

```
def filter_i(p,iterable):
    for i in iterable:
        if p(i):
            yield i
```

Map/Transform, Filter, Reduce/Accumulate:

- Note that Python defines its filter like filter_i: it produces a generator when called.

```
y = filter(predicate.is_prime, irange(2,50))
print(y)
```

prints like <filter object at 0x02C42BF0>.

- Again, we can use a list/tuple/set constructor to turn such a map object into an actual list/tuple/set: it iterates through the values produced by filter and

```
print(list(y))
```

prints

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Map/Transform, Filter, Reduce/Accumulate:

(3) reduce/accumulate: this function is different than the previous two: it takes a binary function and some list/iterable of values and typically produces a single value: it REDUCES or ACCUMULATES its arguments into a single value.

Unlike map and filter (which are defined and automatically imported from the builtins module) we must import reduce from the functools module explicitly.

Once we import reduce, calling

```
reduce(lambda x,y : x+y, irange(1,100)])
```

returns the sum of all the values in the irange iterable. Here is a more interesting call, because uses a non-commutative operator (subtraction).

```
reduce(lambda x,y : x-y, [1,2,3])
```

Map/Transform, Filter, Reduce/Accumulate:

which returns -4: or 1 - 2 - 3 or (1-2)-3. Technically, this is called LEFT reduction/accumulation because the operators are applied left to right. If they had been applied right to left (right reduction), the result would have been 1-(2-3) = 1 - (-1) = 2. For all commutative operators, the association order doesn't make a difference. That is, (1+2)+3 is the same as 1+(2+3). So for 5 values, the reduce is equivalent to (((1+2)+3)+4)+5.

Map/Transform, Filter, Reduce/Accumulate:

- Note that the operator module defines a variety of functions like add (which has the same meaning as lambda x,y: x+y) so we could also call this function as reduce(operator.add, irange(1,100)), if we had imported operator. Other standard Python operators appear in this module as well.
- It also includes the operator.itemgetter, which is a function that takes any number of arguments and returns a function that can be called on any object implementing __getitem__: for example, operator.itemgetter(n) returns a function that can select the nth item in a list: operator.itemgetter(1)([1,2,3]) returns 2, like [1,2,3][1] or [1,2,3].__getitem__(1). Likewise, operator.itemgetter(0,2) returns a function that can select both the 1st and 3rd item in a list: operator.itemgetter(0,2)([1,2,3]) returns (1,3); note that it returns a tuple.

Map/Transform, Filter, Reduce/Accumulate:

- The function attrgetter does the same thing with attributes: operator.attrgetter(attr)(object) returns the same value as either eval('object.'+attr) or object.__dict__[attr]. If c refers to an object with attributes a and b, then calling operator.attrgettr('a','b')(c) returns a tuple equivalent to (c.a, c.b).

Map/Transform, Filter, Reduce/Accumulate:

- Here is another interesting example. Assume that we define a simple max function to return the bigger of its two values.

```
def max(x,y):
    return x if x > y else y
```

- Then, calling

```
reduce(max, [4,2,-3,8,6])
```

is equivalent to max(max(max(max(4,2),-3),8),6) which evaluates (left to right) as follows, to compute the maximum of the entire list of values.

Map/Transform, Filter, Reduce/Accumulate:

```
max(max(max(max(4,2),-3),8),6) -> max(max(max(4,-3),8),6) ->
max(max(4,8),6) -> max(8,6) -> 8
```

Map/Transform, Filter, Reduce/Accumulate:

- Here is a simplified definition of reduce that illustrates its typical behavior on good arguments: it assumes that iterable has at least one value (which it returns, if there are no more values in the iterable).

```
def reduce(f,iterable):
    i = iter(iterable)          # create iterator
    a = next(i)                 # get first value
    while True:                 # while (to process more values in iterator)
        try:                    #   try (to determine if more values)
            a = f(a,next(i))    #     get next/f combines with previous result
        except StopIteration:#   when no more values in iterator
            return a            #     return reduced/accumulated result
```

Map/Transform, Filter, Reduce/Accumulate:

- it is also possible to write this function as follows, since a call to iter(i) when i is already a generator object returns i. Here, the loop "for j in i:" binds j, one at a time, to all the remaining values from i (all values after the first is stored in a).

```
def reduce(f,iterable):
    i = iter(iterable) # create iterator
    a = next(i)        # get first value
    for j in i:        # iterate over all remaining valuse in i
        a = f(a,j) #   using j/f combines with previous result
    return a       # return reduced/accumulated result
```

Map/Transform, Filter, Reduce/Accumulate:

- Hand simulate calls to reduce(max,[1]), reduce(max,[2,1]), reduce(max,[1,2]), etc. to ensure you understand how this code works for 1/2-list, how the loop eventually stops, and how the function computes the correct value. Note that this function raises a **StopIteration** exception if the iterable is empty: the first call to next(i) will raise a **StopException** which is not handled (because it is not in the nested try/except). This is not how Python' reduce works...
- The actual implementation of reduce allows the programmer to specify what value to return for an empty iterable, and is a bit more complicated: e.g., it requires two nested try/except statements. It allows either 2 or 3 arguments: the first must be a binary function (which we will still call f), the second must be an iterable (which we will still call iterable), and the third (if present) is called the unit (aka initializer for the reduction/accumulation).

Map/Transform, Filter, Reduce/Accumulate:

- If the iterable has no values, the unit is returned if it is supplied, otherwise map raises a special **TypeError** (not **StopIteration**) exception. Also, if the unit is specified, it is considered to be the implicit first value in the iterable. All these semantics are captured in the following function, which has the core of the previous reduce, but allows for this new behavior based on the optional third argument.

```
def reduce(f,*args):
    if not (1 <= len(args) <= 2):            # decode the arguments
        if len(args) < 1:
            raise TypeError('reduce expected at least 2 arguments, got '+str(len(args)))
        else:
            raise TypeError('reduce expected at most 3 arguments, got '+str(len(args)))

    iterable = args[0]

    i = iter(iterable)                       # create iterator
    try:                                     # try (to handle empty iterator)
        a = args[1] if len(args) == 2 else next(i)
                                                 # use unit or get first value
        while True:                          # while (to process more values in iterator)
            try:                             #   try (to determine if more values)
                a = f(a,next(i))             #     get next/f combines with previous result
            except StopIteration:            #   when no more values in iterator
                return a                     #     return reduced/accumulated result
    except StopIteration:                    # empty iterator (with no unit): raise exception
        raise TypeError('reduce() of empty sequence with no initial value') from None
```

Map/Transform, Filter, Reduce/Accumulate:

- Note that

```
reduce(operator.add, [])      raises a TypeError Exception
reduce(operator.add, [],  0) returns  0 (the unit)
reduce(operator.add, [5])     returns  5
reduce(operator.add, [5], 0) returns  5 (the unit, 0, + the first value)
reduce(operator.add, [5], 5) returns 10 (the unit, 5, + the first value)
```

- There is only one version of this function, because it produces a single answer, so we don't need separate list/iterable versions. There is a simple recursive definition of the reduce function operating on list (here shown with a mandated unit to keep this function simple), but it uses RIGHT reduction/accumulation, which as we saw for commutative operators produces the same result.

Map/Transform, Filter, Reduce/Accumulate:

```
def right_reduce_l(f,alist,unit):
    if alist = []:
        return unit
    else:
        return f( alist[0], reduce(f,alist[1:],unit) )
```

- Here is a concrete example of a function style of programming, using map, filter, and reduce. This expression computes the length of the longest line in a file.

```
reduce(max,                    # reduce uses max on...
       map(lambda l : len(l.rstrip()),
           # map uses lambda (length of stipped line) on ...
           open('file’))) # an open file
```

Map/Transform, Filter, Reduce/Accumulate:

- So, we use open('file') as the iterable to map, which maps each line to its length; the result of this call is used as the iterable to reduce, which reduces all these line lengths to the single maximum length.
- To return the longest line, not the length of the longest line, we could alter the computation slightly, as follows.

```
reduce(lambda x,y: x if len(x) >= len(y) else y,
       map(lambda l : l.rstrip(),
           open('file')))
```

Map/Transform, Filter, Reduce/Accumulate:

- Here we still use open('file) as the iterable to map, but the lambda to map now just strips spaces off the right end, but doesn't map lines to their lengths.
- The lambda for reduce (whose arguments will be two lines from the file) returns the longer of the two lines by computing and comparing the len of each; when reduced over all lines in the file, the final result is the longest line in the file. If multiple lines are longest, the one appearling earliest will be the result (because of >=).
- Often the form of such a function combines all aspects; one example is:

```
reduce(r_func, map(m_func, filter(f_func, iterable)))
```

Map/Transform, Filter, Reduce/Accumulate:

- which reduces all the mapped values that make it through filter. The functional programming idiom of mapping the result of a filter is already easily captured in Python's comprehension:

```
(m_func(i) for i in iterable if f_func(i))
```

- Functional programmers spend a lot of time using these tools (and writing lots of functions) to build up their idioms of expressions. We are just peeking at this topic now. For example, it is possible for reduce to return all type of results, not just simple ones: there are for example, ways to reduce lists of lists to produce just lists.

Structural Recursion, Accumulation, and Tail Recursion

LECTURE 2

Structural Recursion, Accumulation, and Tail Recursion

- So far, in all the recursive methods that we have written, the form of recursion is based on the data structure the function is processing. Typically, we recur on a substructure (same type of structure but smaller in size: like substrings and sublists) until we reach its base case: the smallest size of that data structure. This is called structural recursion (a more general form or recursion is called generative recursion, which be briefly discuss after this section). A typical example of structural recursion is the list_sum function below.

```
def list_sum (l : [int]) -> int:
    if l == []:
        return 0
    else:
        return l[0] + list_sum(l[1:])
```

Structural Recursion, Accumulation, and Tail Recursion

- A directly recursive function is TAIL-RECURSIVE if the the result the function returns by a recursive call is exactly the value computed by the recursive call, not modified in any way. Notice that the list_sum function is NOT tail-recursive because it returns l[0] plus the result of the recursive call, not just the recursive call.
- We can sometimes transform structurally recursive functions into tail-recursive functions by using an accumulator: an extra parameter that accumulates the answer in each recursive call: typically, the function returns this accumulator in the base case, and the function is tail-recursive because it returns just the result of a recursive call (but with the accumulator updated). Often we define the function by defining and calling a nested function, which has an extra accumulation parameter.

Structural Recursion, Accumulation, and Tail Recursion

- Here is how we can transform the function above into a function using an accumulator, which results in a tail-recursive function.

```
def list_sum ( l : [int] ) -> int:
    def sum_tail(alist,acc)
        if alist == []:
            return acc
        else:
            return sum_tail(alist[1:],alist[0]+acc)
    return sum_tail(l,0)
```

Structural Recursion, Accumulation, and Tail Recursion

- The result of calling the single parameter list_sum(l) function is computed by returning the result of calling the two-parameter sum_tail(l,0) function. The sum_tail function returns the value accumulated in acc when its list parameter is empty; otherwise, is performs structural recursion while increasing acc by the amount of the first value in the list (which is not processed in the recursive call, because it is omitted from the slice). Notice that in the call sum_tail(l,0) the 0 is like the unit in the reduce function: it is the starting point for adding/accumulating values.

Structural Recursion, Accumulation, and Tail Recursion

- Each recursive call adds the front of the remaining values in the list to the sum; when we reach the empty list, all values from the original list have been added into the sum. Here think of the recursion going downward (as illustrated in the hand simulation in class) with the sum incrementing; the bottom function call returns the accumulated result, and each function call above it returns the value returned by its recursive call.

Structural Recursion, Accumulation, and Tail Recursion

- We can transform any tail-recursive function into an iterative one by using a while loop whose test is the opposite of the one for the base case, and whose body updates the parameter names in the same way that they would be updated by performing the recursive function call (binding parameters to arguments); after the body of the loop is a return statement returning the value in acc, the accumulator. We can transform sum_tail as follows.

```
def list_sum ( l : [int] ) -> int:
    def sum_tail(alist,acc):
        while alist != []:
            alist, acc = alist[1:], alist[0] + acc
            # mirrors the recursive call
        return acc
    return sum_tail(l,0)
```

Structural Recursion, Accumulation, and Tail Recursion

- In fact, we can simplify the code for list_sum by replacing the call to the nested function by executing its body prefaced by the initial assignment to its parameters. We can transform list_sum as follows

```
def list_sum (l : [int]) -> int:
    al,acc = l,0   # mirrors the initial call of sum_tail
    while al != []:
        al,acc = al[1:], al[0] + acc
        # mirrors the recursive call to sum_tail
    return acc
```

- All tail-recursive functions can be transformed similarly, running more efficiently (in time and space) than their equivalent recursive functions.

An example of Generative Recursion Example

LECTURE 3

An example of Generative Recursion Example

- Suppose that a country has certain standard stamp denominations. We can represent all the denominations in a tuple. For example, (1,6,14,57), meaning there are 1 cent, 6 cent, 14 cent, and 57 cent stamps for this country.
- Now, suppose that we want to write a function named mns (Minimum Number of Stamps) that computes the minimum number of stamps needed for any amount of postage. The parameters for this function will be the amount of postage and a tuple specifying the stamp denominations that are legal for that country.

An example of Generative Recursion Example

- For example, calling mns(22, (1,6,14,57)) returns 4 (because 22 = 14+6+1+1). You might think to compute this number by using as many of the biggest denomination stamps as you can, then as many of the next biggest denomination stamps as you can, ..., until you have put on enough stamps. But look at 18 cents postage. The approach just mentioned would need 5 stamps: 18 = 14+1+1+1+1, but the minimum number of stamps is 3: 18 = 6+6+6.
- Let's write mns using recursion. We will assume that the amount of postage is initially a non-negative number (and ensure that it is a smaller, non-negative number in all recursive calls). The base case for this problem is postage of 0, which requires 0 stamps.

```
def mns(amount : int, denom : (int)) -> int:
    if amount == 0:
        return 0
    else:
        ....
```

An example of Generative Recursion Example

- For the recursive part, we will try each denomination: reducing the amount by that denomination and computing the minimum number of stamps needed for the choice/reduced amount. Then, we will find which denomination leads to the minimum number of stamps for the reduced amount. The minimum number of stamps needed for the original amount is 1 (for a stamp of the tried denomination) + that number (the minimum). There is one more wrinkle: we cannot use any denomination that is bigger than amount: that would lead to negative postage.
- So, for computing mns(18, (1,6,14,57)) we would compute the following three values by doing recursive calls

```
mns(17, (1,6,14,57))  if we used a  1 cent stamp
mns(12, (1,6,14,57))  if we used a  6 cent stamp
mns( 4, (1,6,14,57))  if we used a 14 cent stamp
```

but not

`mns(-39, (1,6,14,57))` because we cannot use a 57 stamp because it is too big

An example of Generative Recursion Example

- If we computed these values we would compute the following values through various recursive calls that we do not have to think about: it‘s elephants all the way down).

```
mns(17, (1,6,14,57))  returns 3
mns(12, (1,6,14,57))  returns 2
mns( 4, (1,6,14,57))  returns 4
```

- We then compute the minimum of the returned recursive calls, adding one to it for the stamp denomination we used (whether it was 1, 6, or 14 makes no difference when computing the NUMBER of stamps).

```
1 + min( [3, 2, 4] ) which computes 3
```

An example of Generative Recursion Example

- The following code using recursion, iteration, and a comprehension to build a list of the minimum number of stamps needed for amount minus each denomination, if the denomination no bigger than the amount.

```
[mns(amount-d,denom) for d in denom if amount-d >= 0]
```

- It says, for each denomination d, if amount-d >= 0 (meaning d <= amount, so the denomination is legal to use, since the recursive call will have a non-negative -possibly 0- first argument), compute mns(amount-d,denom) and put it in the list. Remember we get to ASSUME that all recursive calls to mns return the correct answer (it's elephants all the way down). We need to take these solutions and solve the original problem, which we do by adding 1 to the minimum number of stamps needed for the smallest solved subproblem.

An example of Generative Recursion Example

- Now we need to compute the minimum of all these values, and return that value plus 1. We can do that by passing the list to the min function.*

```
def mns(amount : int, denom : (int)) -> int:
    if amount == 0:
        return 0
    else:
        return 1 + min( [mns(amount-d,denom)     \
                         for d in denom if amount-d >= 0] )
```

- This solution is quite elegant, although it can take a long time to run for a large amount with many denominations in the tuple. In the next lecture, we will learn how to speed-up this computation using caching/memorization: a decorator that remembers the solution to various problems that are solved over and over again in this recursive solution. This is sometimes referred to as dynamic programming, and is most useful with recursive solutions to problems, when the same small subproblems need to be solved many times (it solves them just once, and remembers the solutions).

An example of Generative Recursion Example

- *Note that calling the min function assumes there is it least one value in the list comprehension. If there is always a 1 cent stamp in the denominations this would be true; if not, then calling mns(1, denominations) would raise an exception, because min would be called on an empty list.

Partial function evaluation

LECTURE 4

Partial function evaluation

- The functools module in Python includes a function named partial that allows us to decorate a function by pre-supplying specified arguments (both positional and keyword); it returns a function that we can call more easily, with fewer arguments: it combines both the actual arguments and the pre-supplied ones to call the function. This kind of decoration is called a partially evaluating a function.
- First let's look at how we can use such a tool and then we will learn how it is written in Python's functools module. We start by defining a simple function that we will partially evaluate, which has two parameters: level and message. It just prints the values of these parameters in a special format.

```
def log(purpose, message):
    print('[{p}]: {m}'.format(p=purpose, m=message) )
```

- Now we will show how to partially evaluate this function using the first and then the second argument.

Partial function evaluation

- Suppose that we want a function named debug to act like log, but have only one argument (message); the argument matching purpose will always be 'debug'. That is, the debug function logs messages whose purpose is always for debugging. We can specify debug as follows, supplying the argument 'debug' to the purpose parameter positionally.

```
from functools import partial
x = 1
debug = partial(log, 'debug’)
# 'debug' is 1st positional argument
```

Partial function evaluation

- Now, calling debug calls the log function always supplying 'debug' as the first positional argument. So, we can call debug with one argument, like

```
debug('Beginning function f')  #call log with 'debug' as 1st positional argument

debug(message = 'x = '+str(x)) #call log with 'debug' as 1st positional argument
```

which prints

```
[debug]: Beginning function f
[debug]: x = 1
```

- Here, calling debug('some message') is like calling log('debug','some message').

Partial function evaluation

- We can also use partial to specify debug as follows, writing purpose = 'debug', which specifies a keyword argument.

debug = partial(log, purpose = 'debug') #'debug' is a keyword argument

- Now, we can call debug with one argument, but it must be supplied as a keyword argument.

```
debug(message = 'Beginning function f') #call log with 'debug' matching purpose
debug(message = 'x = '+str(x))          #call log with 'debug' matching purpose
```

which prints the same as above

```
[debug]: Beginning function f
[debug]: x = 1
```

Partial function evaluation

- Calling debug(message='some message') is like calling log(purpose='debug',message='some message'). Notice that in this case if we called debug with the message positionally, as debug('some message') it would be like calling log('some message',purpose='debug'), which Python would report by raising the TypeError exception, because the first positional argument would match the purpose parameter, and the keyword argument matches the same parameter name. Python would report

- **TypeError**: log() got multiple values for argument 'purpose'

Partially evaluating with the 2nd argument

- Now suppose that we want a function named notify to act like log, but have only one argument (purpose) with the argument matching message always being 'Notify'.
- That is, the notify function logs messages that are the same. We can specify notify as follows, supplying the argument 'Notify' to the message as a keyword parameter only.

```
from functools import partial
x = 1
notify = partial(log, message='Notify’)
#'Notify' is a keyword argument
```

Partial function evaluation

- Now, notify calls the log function always supplying 'Notify' as the first argument. So, we can call notify with one argument, like

```
notify('debug')         # call log with 'Notify' matching message
notify(purpose='log’) # call log with 'Notify' matching message
```

which prints

```
[debug]: Notify
[log]: Notify
```

- Calling notify('some purpose') is like calling log('some purpose',message='Notify') and calling notify(purpose='some purpose’) is like calling log(purpose='some purpose',message='Notify').

Partial function evaluation

- We CANNOT do this kind of partial evaluation using partial(log, 'Notify’) because positionally purpose is the first parameter to log, not message. So that is why there is a difference in partially evaluating the first vs. second parameter to a function.

Two more examples:

- Let's illustrate a few more interesting uses of partial evaluation before showing the Python code that defines this function. It isn't hard to define these two functions conventionally, but it is simple to use the partial function.

(1) Suppose that we wanted a function that returns the index of character in a string of vowels (and -1 if it is not a vowel). We could write.

```
from functools import partial

index_of = partial(str.index,'aeiou')
```

Calling index_of('i') would return 2; calling index_of('z') would return -1.

Two more examples:

- In fact, we can write

```
from functools import partial
index_of = partial('aeiou'.index)
```

and get the same results, because Python actually translates 'aeiou'.index into partial(str.index,'aeiou') by the Fundamental Equation of Object-Oriented Programming: using partial to specify the self parameter.

- Finally, we could also just use a lambda and write

```
index_of = lambda to_check : 'aeiou'.index(to_check)
```

Two more examples:

(2) Suppose that we wanted a function that returns whether or not all characters in a text string argument matches the pre-specified description of an integer with an optional sign. We could write.

```
from functools import partial
import re
is_int = partial(re.match,'[+-]?\d+$')
```

then calling is_int('+33') would return a match object, but is_int('33+') would return None (it does not match).

Two more examples:

- If we wanted to reverse this, and instead write a function that returns whether or not its regular expression argument matches all the characters in a pre-specified text string. We could write

```
from functools import partial
import re
is_match = partial(re.match,string="+33")
```

- Because string is the second parameter, we must specify it as named. Then, calling is_match('[+-]?\d+$') would match (returns a match object) but is_match('\d+$') would not (returns None).

Two more examples:

- All these simple examples use partial evaluation on functions that have just a few arguments. For functions with very many arguments, the usefulness of partial is increased. Here is one more example that uses partial to partially evaluate the print function, which has a few named parameters. Here p_sec is print with sep and end both pre-specified as empty strings.

```
p_seq = partial(print, sep='', end='')
```

calling

```
p_seq(1,2,3)
p_seq(4,5,6)
```

prints: `123456`

again, we could define

```
p_seq = lambda *args : print(*args,sep='',end='')
```

Defining partial as a Python function

- Finally, let's look at how we can simply define (although the code is a bit subtle) the partial function in Python. Here is its defintion and explanation of how it works.

```
def partial(func, *args, **kargs): # bind pre-specified arguments
    def p_func(*pargs, **pkargs):  # bind the arguments in call
        p_kargs = kargs.copy()     # copy kargs dict (from partial)
        p_kargs.update(pkargs)     # update it: add pkargs dict
        return func(*(args + pargs), **p_kargs)
                                   # call the original function
    return p_func                  # return a reference to p_func
```

Defining partial as a Python function

Fundamentally, partial takes arbitrary positional (*args) and keyword (**kargs) arguments; it defines a local function and returns a reference to it. When the local function is called (after partial returns it) it takes the positional arguments (*pargs) it is passed and appends them after the *args passed to partial; it takes the keyword arguments (**pkargs) it is passed and uses them to update the copy of a keyword dictionary that is a copy of the **kargs one originally passed to partial. When it calls func, it takes the appended tupled (*args + *pargs) and expands it to positional arguments and the updated p_kargs dictionary and expands it to keyword arguments.

Defining partial as a Python function

- In fact, the actually definition of partial in Python is more like the following.

```
def partial(func, *args, **kargs): # bind pre-specified arguments
    def p_func(*pargs, **pkargs):  # bind the arguments in call
        p_kargs = kargs.copy()   # copy kargs dict (from partial)
        p_kargs.update(pkargs)   # update it: add pkargs dict
        return func(*(args + pargs), **p_kargs)
                                 # call the original function
    p_func.func      = func      # Remember (in a queryable form)
    p_func.args      = args      #   all arguments to partial:
    p_func.keywords  = kargs     #   func, args, and kargs
    return p_func                # return a reference to p_func
```

Defining partial as a Python function

- In this version we add three attribute names to the p_func function object that is returned, recording useful information for it, which we can query. Function objects, like all other object, can have attributes. So, if we wrote

```
debug = partial(log, "debug")
print(debug.func, debug.args, debug.keywords)
```

- Python would print:

```
<function log at 0x02CCA780> ('debug',) {}
```

Defining partial as a Python function

- Showing the actual function being called, and what positional and keyword arguments are automatically going to be suppplied.
- In a simpler context, we can define

```
def f(x):
    return f.mult * x
```

and then we can later set `f.mult = 2`, so calling `f(3)` returns `6`. So, function objects have name spaces that we can manipulate as well.

MapReduce, associative functions, and parallel processing

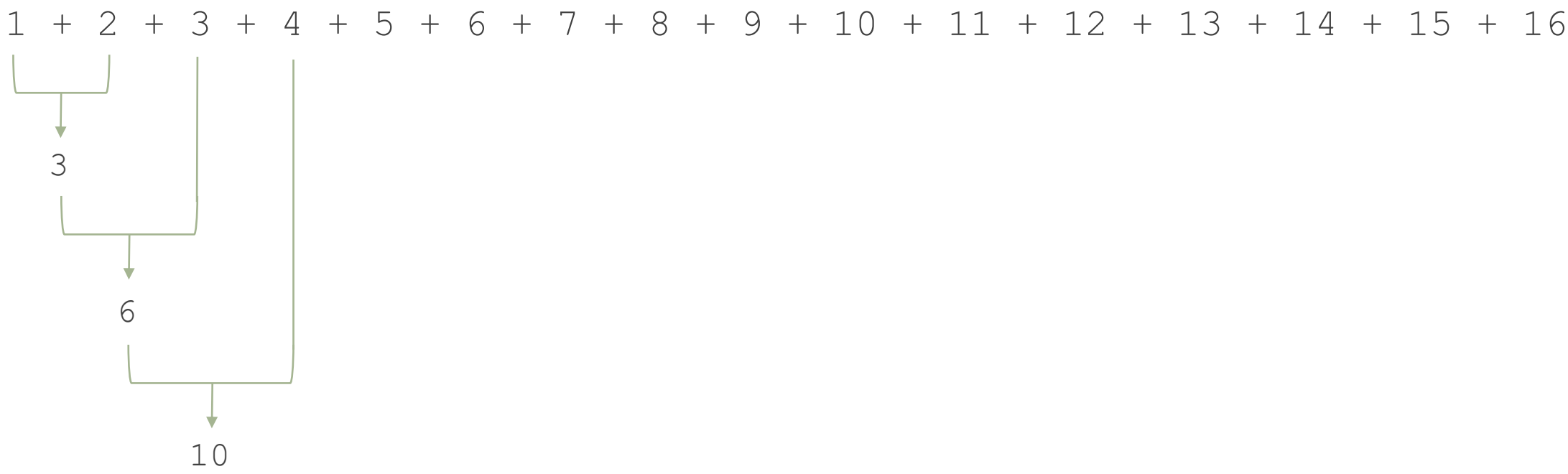
LECTURE 5

MapReduce, associative functions, and parallel processing

- MapReduce is a special implemention of the map/reduce functions implemented to run in parallel on cluster, or multi-core computers. If we can write our code within the MapReduce language, we can guarantee that it runs quickly on the kinds of computers Google uses for its servers. Generally, what it does is run similar operations on huge amounts of data (the map part), combining results (the reduce part), until we get a single answer. Apache Hadoop is open source version of MapReduce (but to really see its power -the decreased execution time-, we need huge amounts of data and a cluster of computer to run our code on.

MapReduce, associative functions, and parallel processing

- How does MapReduce work? The story is long, but here is a quick overview. Imagine we have an associative operator and want to compute: 1 + 2 + 3 + ... + n
- We can evaluate this expression as shown above, which would require n-1 additions one right after the other (the former must finish before the later starts). Even if we had multiples cores, doing the operations in this order would require n-1 sequential additions because only one core at a time would be active.

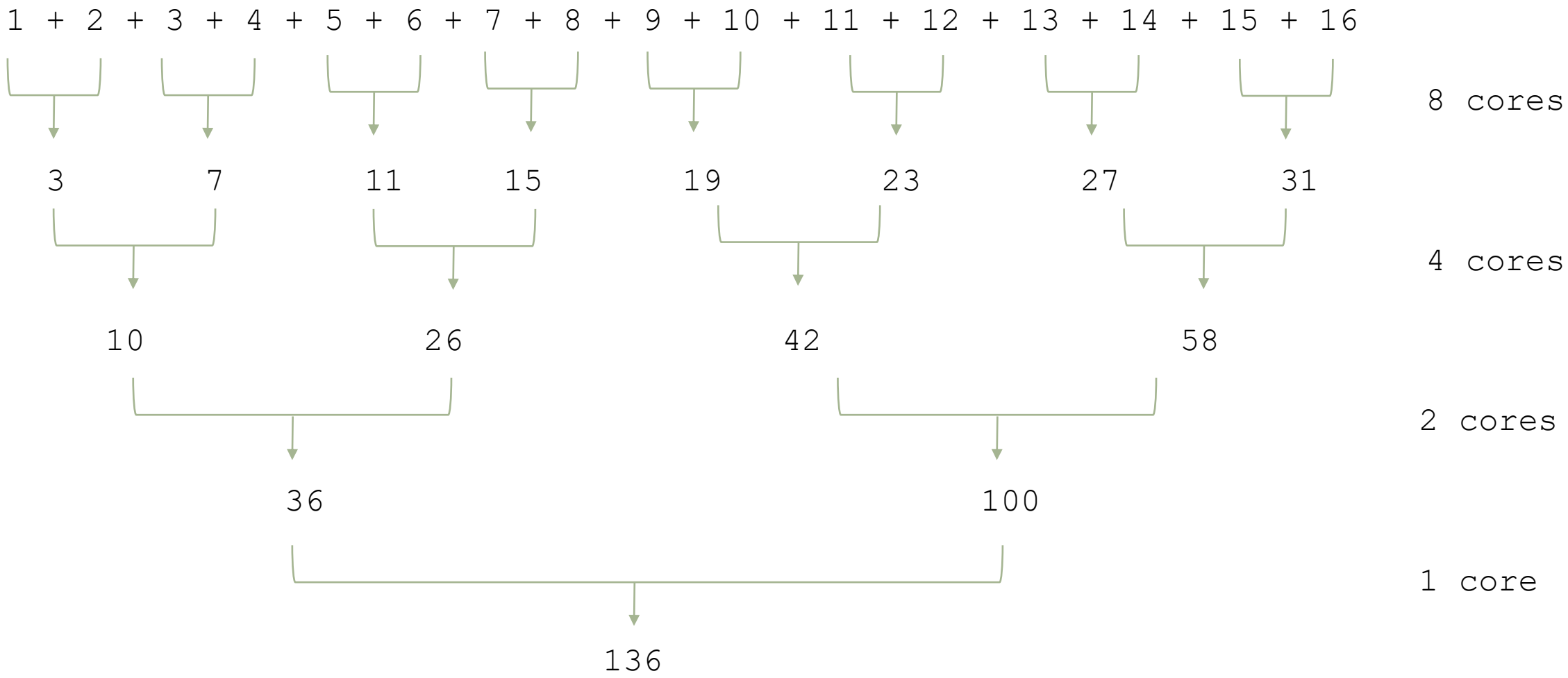


…. note that one more operand is used at each level

MapReduce, associative functions, and parallel processing

- Here each level uses 1 core and there are 15 levels. In general, with N numbers to add it take N-1 time steps/levels.
- Now, how MapReduce can handle this problem?
- Instead, because of associativity, we can evaluate this expression in a different way: add the 1st and 2nd values at the same time as we add the 3rd and 4th at the same time as the 5th and 6th ... In all, we can add n/2 pairs simultaneously (if we had n/2 cores). We can use this same trick for the remaining n/2 sums, simultaneously adding them together; then for the n/4 sums,

..., to the final sum sums (for which only one processor is necessary). Here is a pictorial representation of this process for 16 values.


1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16
8 cores
3 7 11 15 19 23 27 31
4 cores
10 26 42 58
2 cores
36 100
1 core
136

MapReduce, associative functions, and parallel processing

- Here each level uses as many cores as possible there are 4 levels. In general, with N numbers to add it takes Log2 N times steps. Recall that Log2 1,000 is 10, Log2 1,000,000 is 20, and Log2 1,000,000,000 = 30. But, to perform 10**9 additions in 30-time steps, we'd need a half billion cores: not likely this is coming in your lifetime. But if we had tens-or-hundreds of cores, we could keep them all busy except for the last few (bottom) levels. So, we could get our code to run tens-or-hundreds of times faster. Of course, the unneeded cores can start working on the next MapReduce problem.

Combinatorial Computing

LECTURE 6

Combinatorial Computing

- This lecture got too long for this section to be detailed, which mostly was going to be about various iterators in the itertools module. Instead, I have listed just the combinatoric generators there. Here is just a quick overview of some intuitive and useful ones. The itertools module covers other interesting iterator decorators to peform a wide range of operations.

product(*iterables)

- produces tuples with the cartesian product of the iterators, where each represents one "dimension). For example list(product('ab',irange(1,3))) produces the following list of 2-tuples

```
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3)]
```

Combinatorial Computing

permutations(iterable,r=None)

- produces tuples (in sorted order) that are permutations of the values produced from iterable. If r is specified, each result is an r-tuple (if not, each tuple has all the values in the iterable). For example list(permutations('abc')) produces the following list of 3-tuples of 'abc'

```
[('a','b','c'),('a','c','b'),('b','a','c'),
 ('b','c','a'),('c','a','b'),('c','b','a')]
```

list(permutations('abc',2)) produces the following list of 2-tuples of 'abc'

```
[('a','b'),('a','c'),('b', 'a'),
 ('b','c'),('c','a'),('c', 'b')]
```

- Generally, if iterable has n values, the number of tuples returned is n!/(n-r!) when 0<=r<=n and 0 when r > n.

Combinatorial Computing

combinations(iterable,r)

- produces r-tuples (in sorted order) that are combinations of the unique values produced the from iterable (where only the values, not their order, is important). For example, list(combinations('abcd',3)) produces the following list of 3-tuples of 'abcd'

```
[('a','b','c'),('a','b','d'),('a','c','d'),('b','c','d')]
```

- Generally, if iterable has n values, the number of tuples returned is n!/r!(n-r!) when 0<=r<=n and 0 when r > n.

Combinatorial Computing

combinations_with_replacement(iterable,r)

- produces r-tuples (in sorted order) that are combinations of the values (which don't have to be unique) produced the from iterable (where only the values, not their order, is important). For example, list(combinations_with_replacement('abc'),2) produces the following list of 2-tuples of 'abc'

```
[('a','a'),('a','b'),('a','c'),
 ('b', 'b'),('b','c'),('c','c')]
```

- Generally, if iterable has n values, the number of tuples returned is (n+r-1)!/r!(n-r!) when n > 0