This document is intended as an introduction to Python for people with prior programming experience. Emacs (with Python mode) is **highly** recommended for editing your code. We will also assume that you are familiar with emacs—if not, you can find a variety of excellent tutorials and references on the internet. You can find more Python-related Emacs packages at `https://www.emacswiki.org/emacs/PythonProgrammingInEmacs`

Contents

1 Getting Started

Python does not have a static type system. Instead, if you use the wrong type of data in an operation, you will get a run-time error. This is most unfortunate, as it means that type errors must be caught by test-cases and debugged, rather than being directly identified by the compiler.

For small interactions, you can evaluate code at the Python read-eval-print-loop (REPL). To do this, run python3 (or python for version 2), and you will be presented with a prompt like this:

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can then evaluate Python code at the prompt, and it will print the resulting value for you. For example:

```
>>> 3+4*7
31
>>> "Hello World"
'Hello World'
>>> 'Another String'
'Another String'
>>> [1,2,3]
[1, 2, 3]
>>> (1,2,3)
(1, 2, 3)
>>> [(1+2,3), 4,7,9,"hello"]
[(3, 3), 4, 7, 9, 'hello']
```

In the first line of this interaction, the user evaluates the expression `3+4*7`, and Python reports that it evaluates to 31. The next two lines show strings. Python accepts either double or single quotes for string literals. Single quotes are more common among Python programmer, but double quotes are most common in other languages. This example also shows a list (`[1,2,3]`), and a tuple (`(1,2,3)`), groups multiple values into one value (this is most similar to `std::tuple` in C++11, which generalizes `std::pair` to have an arbitrary number of components).

Note that Python's lack of a static type system allows a list that would be ill-typed in strongly typed languages. The last list evaluated contains three different types of data, and thus is not well-typed in most type systems[1].

If the programmer is writing a larger piece of code, however, she likely does not want to type it all at the REPL. Instead, she would save it in a file (typically with the `.py` extension). While editing in Emacs (in Python mode), the contents of the entire current buffer can be sent to Python with C-c C-c (if Emacs is not already running Python, it will prompt you for how to run it before sending the buffer). You can also send the currently selected region with C-c C-r. Note that this works extremely well with C-M-h, which selects the current function definition (so if you want to send one function, C-M-h, then C-c C-r).

We also note that Python has a debugger. See `https://docs.python.org/3.5/library/pdb.html` for details.

2 Conditionals, Loops

Unlike many programming languages which use matching punctuation (*e.g.*, `{}`) to indicate blocks and nesting, Python uses indentation. For example, the following code defines a function (`fizzbang`), which has a for loop inside of it. Nested inside of that loop, there is an `if/elif/else` statement, and inside each clause of that conditional, there is a `print statement`:

```
def fizzbang(n):
    for i in range(n):
        if (i % 15 == 0):
            print ('fizzbang')
        elif (i % 3 == 0):
            print ('fizz')
        elif (i % 5 == 0):
            print ('bang')
        else:
            print (i)
            pass
        pass
    print ('Can I get a serious question now?')
    pass
```

Notice that the *only* indication of how these statements nest inside of each other is their indentation. One thing which may seem a bit odd here is the `pass` statements. In Python, `pass` does nothing. It can be used in places where a block of code is required, but you want to do nothing. However, a more common use is to tell your editor that you have reached the end of a nesting (where you would normally close a curly brace). Emacs recognizes `pass` as meaning "the next line should be indented one level less, and thus back out a level of nesting").

[1] A reader not versed in type theory may think that this adds "power" to the language relative to a strongly typed language. However, a well-designed type system would let the programmer accomplish a similar computation in a type safe way: see datatypes and pattern matching in SML for an excellent example.

2.1 Conditionals

Now that you have seen nesting, conditionals are pretty straightforward, even though their syntax is different from most other languages. The condition is followed by a colon, then its body is nested inside by indenting it another level. Python also has "elif" for "else if".

Python does *not* have `switch case` (or an equivalent). You are instead either expected to make a series of `if elif...elif else` clauses, or to look up each case in a dictionary (which we will discuss later).

One thing that is a bit unusual about Python is that it allows *any* type to be used as a conditional expression. When used in a context where a boolean is required, most things are treated as true, however, the following things are treated as false:

- False
- None (which is used to indicate the absence of a value)
- Numeric values which are 0
- Objects on which `__bool__()` is defined, and it returns `false`.
- Objects on which `__bool__()` is not defined, but `__len__()` is defined, and `__len__()` returns 0.

Note that this last rule means that empty sequences (such as the empty list, an empty dictionary, an empty set, etc), are all treated as false.

Python has the typical (in-)equality operators (`<`, `<=`, `>`, `>=`, `==`, and `!=`), as well as `not`, `and`, and `or`—the later two of which are short-circuiting (just like `&&` and `||` in C/C++/Java). However, unlike most languages, multiple inequality operations can be written together (`a < b <= c < d`) as syntactic sugar for `and`-ing each operation together (`a < b and b <= c and c < d`). Note that since this is syntactic sugar for `and` (which short-circuits), the original inequality is also short-circuiting (if `a>=b`, then `b<=c and c<d` will not be evaluated at all.

Python also has `is`, `is not` and `in`. The first two of these are for checking if one object is exactly the same object as another object (Think of the Python `a is b` as being the C `&a == &b`). Likewise, `a is not b` checks if the objects are distinct (in C: `&a != &b`). The third of these, `in`, is used to check if an item is in a sequence. For example `thing in somelist` checks if the value of `thing` occurs in `somelist`. You can also use `in` to check if one string is a substring of another.

2.2 Loops

For loops are typically written in what is called "for each" style. In Python, this is `for var in expr:`. The expression should evaluate to something that is *iterable*. Many things in Python are iterable: lists, strings, and dictionaries. You can make your own classes iterable by defining the `__iter__` method in them. Python also supports `while` loops. `break` and `continue` work in Python as you would expect.

One unusual feature of `for` loops in Python is that they can have a `else` clause. An `else` clause of a `for` loop gives a chance to execute code that only happens if the loop iterated over all items—as opposed to exiting due to a `break` statement. For example, we could make our `fizzbang` a bit more general:

```
def fancyfizzbang(n, replacements):
    for i in range(n):
        for mod,repl in replacements:
            if (i % mod == 0):
                print(repl)
                break
            pass
        else:
            print(i)
        pass
    pass
```

This function takes a list of pairs (`replacements`), and for each number in the range, iterates over that list. If the inner loop finds a divisor in the list, it breaks. This means that it does NOT execute the `else` clause. If it finishes iterating over replacements without breaking, it performs the else (and prints i). Note that the `else` is at the same indentation level as the `for`, NOT the same indentation level as the `if`. You could call it like this:

```
fancyfizzbang(100,[(15,'fizzbang'),(5,'bang'),(3,'fizz')])
```

Note that this function uses tuples (here pairs), which make one value out of many (kind of like `std::pair`). You write them with parentheses around comma-separated values. When you assign a tuple to something, you can split it apart by writing comma-separated names to bind each part to. In the example above, `mod` and `repl` are two variables which are bound to the first and second part of the pair (*e.g.*, 15 and `'fizzbang'` in the first iteration). Note that square brackets around a comma-separated sequence of values is a list, as you saw earlier.

3 Data Structures

We briefly mentioned tuples above, but these, and other important data structures merit a bit further discussion before we proceed.

3.1 Sequences: Tuples, Lists, and Strings

As we noted previously, a tuple is written as a comma-separated sequence of values inside parenthesis. Tuples are *immutable*, meaning you cannot change them once you create them. Unlike many other languages, Python tuples support iteration over their elements. That is, we can write something like this:

```
for i in (1,5,'hello',2):
    print(i)
```

As you saw previously, tuples can be unpacked, by assigning to multiple variables (one per tuple element). For example, we could evaluate the following code

```
a = (1, (2,3), (4,5,6),((7,8),9))
b,c,d,e =a
print('b= ' + str(b) + ' c= ' + str(c) + ' d= ' + str(d) + ' e=' + str(e))
x,y,z=d
print('x= ' + str(x) + ' y= ' + str(y) + ' z= ' + str(z) )
m,n=e
print('m= ' + str(m) + ' n= ' + str(n) )
```

and get this output:

```
b= 1 c= (2, 3) d= (4, 5, 6) e=((7, 8), 9)
x= 4 y= 5 z= 6
m= (7, 8) n= 9
```

Python also has *lists*. A list is also a sequence of data. One major difference between a list and a tuple is that a list is *mutable*, meaning that it can be changed. Lists typically have a connotation of all elements having the same type, whereas tuples do not (languages with static type systems require all elements of a list to be the same type, while tuples may be heterogeneous—Python does not have a static type system, so it will not stop you from putting different types in a list).

There is a special `range` type for ranges of integers. They are generally made with `range(start, end)`, but can also be done with `range(start, end, step)`. These have a more space efficient representation than a list containing all the elements in the range. Note that ranges are half-open intervals (`[start,end)`).

For API documentation on tuples, lists, and ranges.See:
`https://docs.python.org/3.6/library/stdtypes.html#sequence-types-list-tuple-range`

A *string* is an immutable sequence of characters (which, as you have seen before, can be written literally with either single or double quotes). Many other types of data can be converted to a string representation by `str(whatever)`, for example,

```
>>> str(3)
'3'
```

See `https://docs.python.org/3.6/library/stdtypes.html#text-sequence-type-str` for API reference on strings.

3.2 Indexing and Slicing

From other programming languages, you are likely familiar with indexing. For example, if you wrote

```
x=[1,2,3,4,5,6,7,8,9,10]
x[0]
```

then you would expect `x[0]` to evaluate to 1, as the `[0]` indexes the list, obtaining its first element (as in most other programming languages, indexes start at 0 in Python). Note that you can index any of the sequences we just discussed: lists, tuples, and strings.

Python also support list "slicing" with syntax similar to indexing—you can write `[start:end]` after a sequence value to get the elements in the range `[start,end)`. If either the start or the end indices are omitted (but the colon is included), the slice goes to the start/end respectively. For example, given the above initialization of `x`, we could evaluate the following:

```
>>> x[3:5]
[4, 5]
>>> x[3:]
[4, 5, 6, 7, 8, 9, 10]
>>> x[:3]
[1, 2, 3]
```

You may also specify a third number in the square brackets to specify a stride in the elements to select. For example, `x[3:8:2]` gets every other element in the range `[3,8)` from `x` (which would be the list `[4,6,8]` if `x` is the list shown above).

3.3 Comprehensions

Python supports building of sequences by *comprehensions*—expressions which describe how to build the sequence. For example, we might write the following:

```
>>> [ 3*x+2 for x in range(0,20) if x % 3 == 2]
[8, 17, 26, 35, 44, 53]
```

What exactly happened here? First of all, `range(0,20)` generates a sequence of the integers in the range `[0,20)`. The `if x % 3 == 2` filters that sequence down to the ones that meet the specified condition (here that the number mod 3 is 2—so the list `[2, 5, 8, 11, 14, 17]`). Then the list is built by taking `3*x2+` for each of those numbers (so 3*2+2 = 8, 3*5+2 = 17, and so on). Most generally, a list comprehension will have the form:

```
[ EXPR for VAR in SEQ if COND ]
```

Where EXPR is the expression to evaluate for each item in the list, VAR is the variable to use to represent each item, SEQ is an expression which generates some other sequence (to bind each value of VAR to), and the `if COND` is an optional clause which filters the values of SEQ before evaluating them.

3.4 Sets

Python also has built in sets, which can be written literally with `{}`. Sets work as you would expect from learning about set ADTs in your prior programming experience: they are *unordered* collections, which support the mathematical set operations (membership testing, union, intersection, difference,subset, and testing for equality). Note that since the sets are unordered, `{1,2,3}=={2,3,1}` is true.

As with the other types we have seen, you can write set comprehensions in Python, such as

```
{x for x in range(0,20)}
{str(x) for x in [1,2,3,4]}
```

See `https://docs.python.org/3.6/library/stdtypes.html#set` for API reference.

3.5 Dictionaries

One of the most ubiquitous data structures in Python is the "dictionary" (which is called a *map* in most other languages). Python's dictionary supports what you would expect from a map ADT: adding key/value pairings (`d[k]=v`), and looking up values from their key (`d[k]`). Dictionaries can be written literally (`{k1:v1, k2:v2}`) and may be iterated over.

You can also write dictionary comprehensions, which look much like set comprehensions, except that instead of having a single expression before the `for`, you have *keyexpr* : *valexpr*. For example:

```
{x:y for x,y in [(1,2),(3,4)]}
```

builds the dictionary that maps 1 to 2 and maps 3 to 4. This example is not terribly exciting, but suppose you had two lists (klist and vlist) of equal length, and wanted to build a dictionary mapping each element of klist to the corresponding element of vlist:

```
{k:v for k,v in zip(klist,vlist) }
```

See `https://docs.python.org/3.6/library/stdtypes.html#mapping-types-dict` for API reference on dictionaries.

4 Exercises 1

At this point, you are ready to try a few things of your own:

1. Write a set comprehension to generate all the odd numbers between 1 and 355 (inclusive of both).
2. Write a list comprehension which takes the numbers in your previous set, and for each one that is congruent to 1 mod 3, puts the square of that number in the list.
3. Write a for loop which iterates over the elements of the list comprehension that you just made and prints each "Element (n): (v)" where (n) is replaced with the index of the element, and (v) is replaced with the value of the element, one element per line for each element. (So the first line should be "Element 0: 1").

5 Functions

You have already seen the definition for a couple functions in previous examples. Functions in Python follow many of the same principles that you are used to from other programming languages: they take parameters, perform some computation, and return a value. If a function does not explicitly return a value, it implicitly returns `None`, which (as mentioned earlier) indicates the lack of a value. Python has a variety of useful features related to functions.

5.1 Higher-Order Functions

In Python (as in functional languages), functions can be passed as parameters to other functions, returned from functions, placed in data structures, and generally treated like any other object. Let's see one of these in action. Suppose you have defined this simple function:

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

and evaluate the following:

```
>>> list(map(square,[1,2,3,4,5]))
[1, 4, 9, 16, 25]
```

Notice that we passed the *square function* as the second argument of map. We convert the return value of map to a list, as we are using Python3 (in Python 2, it returns a list—in Python3 it returns something a bit more complex, which we will discuss later). Note that unlike a strongly typed language where `map` would typically require a list, Python's map will accept any type of iterable: lists, sets, strings, etc.

There is nothing magical about `map`: we can write our own implementation of it. Here is one that matches with Python2's behavior:

```
def myMap(f,lst):
    ans=[]
    for x in lst:
        ans.append(f(x))
        pass
    return ans
```

Notice that `myMap` takes a parameter (`f`) which is a function, and calls it on each element of `lst` (`f(x)`).

Note that Python purists generally prefer list comprehensions to `map`.

5.1.1 Lambdas

When using higher-order functions frequently, it is rather cumbersome to write out many small functions explicitly. Instead, we might prefer to just write an anonymous function where we want to use it. For example, we might want to do:

```
>>> myMap(lambda x: x*x,[4,6,9])
[16, 36, 81]
```

Here, we wrote `lambda x: x*x` to make an anonymous function. In general, the syntax is `lambda` *params* `:` *body*.

5.2 Filter and Reduce

Two other really useful/common higher-order functions are `filter` and `reduce`. Filter takes a function which decides which elements should be kept in the resulting answer, and an iterable to filter. The result has the elements for which the function returned true. For example:

```
>>> list(filter(lambda x: x > 7, [1,4,8,9,2,12,3]))
[8, 9, 12]
```

Here, the elements from the input list which are larger than 7 are kept in the output list. As with `map`, Python purists generally prefer comprehensions to `filter` (*e.g.,* `[x for x in [1,4,8,9,2,12,3] if x > 7]`).

The `reduce` function combines all elements from a list (or other iterable) using some function which takes a current answer, an element from the list, and returns a new answer (basically, `fold` from most functional languages). An initial answer is optional—if it is not provided, the first element of the list is used. If you want to use reduce, you need to import it:

```
from functools import reduce
```

Then, we could use it to write a function which sums a list (or other iterable whose elements can be added):

```
def sumList(lst):
    return reduce(lambda x,y: x+y, lst)
```

If we call `sumLst([1,2,3])` then it will call our lambda on 1 and 2 (adding them to get 3), then call our lambda again on 3 and 3 to get 6, which is the answer.

You can use reduce a bit more generally by passing an optional third argument (we'll talk about optional arguments shortly), which is the initial value. For example, if we wanted to reverse a list (Python purists would hate this, SML programmers would approve of the functional nature, but be annoyed at other aspects):

```
def revList(lst):
    return reduce(lambda x,y: x.insert(0,y) or x, lst,[])
```

Now if we do `revList([1,2,3])`, reduce will call our lambda with `x=[]`, and `y=1`. This will insert `1` at the start of `x`, and then evaluate to `x`. Note that we can't put multiple statements in the `lambda`, so we abuse `or` here, since `insert` returns `None` which is false when evaluated in a boolean context. Then reduce calls our lambda with `x=[1]` and `y=2`, inserting 2 at the start of the list to form `[2,1]`. One more call of the lambda results in `[3,2,1]`.

5.3 Parameters

Python has a handful of features in parameter passing. One of these is the ability to supply a default value for a parameter, allowing the caller to omit that parameter if the default value is desired. You saw this previously with `reduce` which could be passed 2 or 3 parameters: the third parameter had a default value in the definition of reduce.

Providing a default value for a parameter is accomplished by writing `=value` for whatever value you want to use, right after the parameter. For example:

```
def myFn(a, b=3, c='hello'):
    print ('a=' + str(a))
    print ('b=' + str(b))
    print ('c=' + str(c))
```

Here, `a` has no default value, and must be passed to `myFn` explicitly. The other two parameters have default values (`3` and `'hello'` respectively). Accordingly, any of the following calls are legal:

```
>>> myFn(1)
a=1
b=3
c=hello
>>> myFn(1,4)
a=1
b=4
c=hello
>>> myFn(1,4,'world')
a=1
b=4
c=world
```

Note that we could pass in different types than the default value (since there is no static type system).

```
>>> myFn(1,'xyz',42)
a=1
b=xyz
c=42
```

5.4 Variable Positional Arguments

Python allows you to write functions that take any number of arguments by naming a parameter starting with a single `*` (conventionally called `*args`, but any name is allowed). For example, suppose we wanted to write a function that sums its parameters, but can take an number of parameters.

We could write

```
def manySum(*args):
    ans=0
    for i in args:
        ans+=i
        pass
    return ans
```

We can now call `manySum` with any number of parameters (even 0):

```
>>> manySum(1,2,3)
6
>>> manySum(1,2,3,5,8,9)
28
>>> manySum()
0
```

You can use `*args` with other parameters before it. In such a case, the other parameters are required, and 0 or more other parameters may be passed as `args`. For example, you could write:

```
def manyF(f,ans,*args):
    for i in args:
        ans=f(ans,i)
        pass
    return ans
```

Here, `f` and `ans` are required, but then we may pass 0 or more other parameters to be `args`:

```
>>> manyF(lambda x,y: x*y,1, 4,5,6,7)
840
>>> manyF(lambda x,y: x-y+3, 42, 9, 8, 3, 2, 7, 12)
19
>>> manyF(lambda x,y: x*y,1)
1
```

5.5 Keyword Parameters

Python also allows parameters to be passed by keyword, that is with name=value in the function call. This syntax is useful when you want default values for some parameters, but to pass non-default values for later parameters. It is also useful for readability when it might be hard to tell which parameter is which at the call site.

For example, suppose we had a function definition such as:

```
def f(x=0,y=True,z=42):
#we dont care about the body for this example
```

Now, suppose we wanted to call f using the default values for `x` and `z` but pass in another value (*e.g*, False) for `y`. If you just wrote `f(False)`, then False would be passed for `x`. However, you can write `f(y=False)` to accomplish what you want.

If you want to take arbitrary parameters as keyword parameters, you can do so with `**kwargs`. When you do so, kwargs is a dictionary mapping parameter names to parameter values. For example, we could define the following function:

```
def prArgs(**kwargs):
    for k in kwargs:
        print(k+ "=" + str(kwargs[k]))
        pass
    pass
```

This function accepts any number of keyword arguments, and prints them out (by iterating over kwargs to get the keys, then lookup them up in kwargs).

5.6 Documentation

In Python, the correct way to document a function is to write a "Docstring" as the first line of the function. This Docstring should be a string written with three sets of quotation marks (`"""`) at the start and end. If it is longer than one line, it should have a short one line description, followed by a blank line, followed by a more detailed description. Following these rules allows your documentation to be processed by various tools that work with Python documentation.

For example, we might document our fancy fizz bang function with a Docstring:

```
def fancyfizzbang(n, replacements):
  """This function implements the 'fizzbang' algorithm with arbitrary replacements

  Parameters:
  n (int) the ending number of the range over which to fizzbang
  replacements (sequence of (int,string) tuples) the set of replacements to use
               each number in the range is tested against the divsiors (integers)
               from this set).  If the number is a multiple of the divisor, it
               is replaced with the corresponding string from that tuple
  """
  for i in range(n):
     #...remainder of code omitted...
```

Note that Docstrings can (and should) also be placed at the start of modules, classes, and methods.

Docstrings differ from normal comments (which are done with `#` to comment to end of line) in that they are no ignored by the compiler, but rather incorporated as an element called `__doc__` in the entity that they are documenting (if you define `fancyfizzbang` with a Docstring, and write `fancyfizzbang.__doc__`, you will see the string above).

5.7 Decorators

One incredibly nice feature of Python is the idea of "decorators". Decorators are higher-order functions, which take a function as a parameter and return a function which "wraps" the passed-in function to modify its behavior. By "wraps" the passed-in function, we mean that the modified function can act before the target function is called (*e.g.*,checking and/or modifying its parameter values) or after it returns (*e.g.*, examining the return value). Decorators can have a wide variety of uses—logging, profiling (timing how long a function takes), or enforcing preconditions (checking that parameters values meet certain requirements)—just to name a few.

We will start with a simple example—a decorator that times how long another function call takes. We would want to do a few other things (which we will see shortly) when making a real decorator.

```
def profile(f):
    def wrapper(*args,**kwargs):
        start=time.clock()
        ans=f(*args,**kwargs)
        end=time.clock()
        print("Function call took " + str(end - start) + " seconds")
        return ans
    return wrapper
```

Take a moment to break this function down. The function `profile` takes another function (`f`) as its parameter. It then defines *another* function (`wrapper`) inside of itself. This wrapper function takes an arbitrary set of positional and keyword parameters: we want it to work with any other function, f, no matter what parameters f expects. The wrapper calls `time.clock()` to get the starting time (we assume that we previously did `import time`). It then calls `f` with whatever

arguments it was passed, and remembers the return value of `f`. Next, it calls `time.clock()` again to find out when the function ended, prints out a message about how long the function took, and then returns whatever `f` returned.

Note that `profile` itself does not actually execute a call to wrapper, it simply returns the *function* `wrapper` itself.

Now, before we discuss how to use this as a decorator, let us just take a moment to note that there is not really much here that we could not have done in another functional language. For example, in SML we could have written `profile`, and though the syntax would be different, it would still have the same effect: a function which takes in a function and returns another function with the same behavior, but also prints out the time it took. In fact, we note that SML can even do this in a strongly-typed way (`profile` would have type `('a -> 'b) -> 'a -> 'b`) without unduly restricting its behavior. We could even use `profile` in the same way that we would in some other functional language:

```
timedFizzbang=profile(fizzbang)
timedFizzbang(600)
```

This gives us some nice functionality, but Python's decorators add syntactic simplicity by letting us modify the function's behavior at the site of its definition. We can simply write `@profile` before the function definition of a function, and that function will be wrapped in the decorator. That is, if we wanted to always profile our calls to fizzbang, when we define fizzbang, we write

```
@profile
def fizzbang(n):
    for i in range(n):
    #...remainder of code elided..
```

Now that you have seen the use of a decorator, we will make two small improvements to our profiling function:

```
def profile(f):
    def fToString(*args, **kwargs):
        ans=f.__name__
        ans=ans+"("
        sep=""
        for a in args:
            ans=ans+sep+str(a)
            sep=","
            pass
        for k in kwargs:
            ans=ans+sep+str(k)+"="+str(kwargs[k])
            sep=","
            pass
        ans=ans+")"
        return ans
    @wraps(f)
    def wrapper(*args,**kwargs):
```

```
        start=time.clock()
        ans=f(*args,**kwargs)
        end=time.clock()
        print(fToString(*args,**kwargs) + " took " + str(end - start) + " seconds")
        return ans
    return wrapper
```

The first improvement that we made is that instead of just printing "Function call took" we actually print the name and arguments passed to the function, so that we can tell which call is which. The second improvement is that we decorated `wrapper` with the `@wraps(f)` decorator. Note that unlike our use of `@profile`, we are passing an argument to `@wraps` (namely `f`). This is totally fine, as long as our decorator is written to take parameters. Such a decorator is written by declaring a function which takes the desired parameters and returns the decorator itself.

What exactly does the `@wraps` decorator do? It copies the metadata from the function which is its argument (in this case, the parameter `f`) to the function it decorates. This metadata includes things like the function's name and Docstring. Without it, examining the decorated function's name (`fizzbang.__name__`) would just give us the string "wrapper"). However, with the `@wraps` decorator, `fizzbang.__name__` gives us "fizzbang" as we would expect.

6 Exercises 2

1. Write `fact(n)` which computes factorial of `n` recursively.

2. Write `reduceFact(n)` which computes factorial of `n` by using `reduce` and `range`. Your implementation should use neither recursion nor iteration explicitly. Instead, it should generate an appropriate range of numbers and `reduce` them to compute its answer.

3. Decorate your recursive `fact` with the `profile` decorator above. Evaluate `fact(6)`. Did it do something you did not expect? Why do you think this happened?

4. Modify your recursive `fact` function so that it remedies the unexpected/undesired behavior you identified in the previous problem. Be sure that you still use recursion (and not iteration) in your new solution. Hint: you can write one function inside another.

5. Write a decorator `log`. The decorator should take two arguments, `before` and `after` which are messages to print as the function is called (before), and as it returns (after). When the decorated function is called, it should print the "before" message, followed by the function named and its arguments (as with `fToString` above). Then it should call the function remembering the return result. It should then print the "after", along with the function name and its arguments, followed by the return result. Finally, this function should return the result of the function it decorates. For example, I should be able to do

   ```
   @log(before='Calling ', after='Returning from ')
   def fact(n):
     # whatever you wrote previously
   ```

   and get an output of

```
Calling fact(6)
Returning from fact(6) = 720
```

(as well as having 720 returned).

6. Modify your `log` decorator so that the `before` and `after` are treated as a string with `$` replacements : `$name` should be converted to the name of the function `$args` should be converted to the arguments passed to the function, and `$result` should be converted to the return result (for *after*) only. Give a default value for `before` of `"Calling $name($args)"` and a default value for `after` of `"Returning from $name($args) = $result"`. Test your new decorator. (Hint: look up the API documentation for "Template Strings").

7 Objects

Python supports classes, objects, and inheritance. In fact, everything in Python is an object. We have already alluded to the fact that functions are objects, by doing `.__name__` on them in our discussion of decorators.

The first thing we for our discussion of OOP is a class definition:

```
class Point:
    def __init__(self, x, y):
        self.x=x
        self.y=y
        pass
    pass
```

This defines a Point class, with a constructor. In Python, the constructor is called `__init__`, and the `this` pointer is called `self` and is passed explicitly. The constructor takes two other parameters (`x` and `y`) and initializes fields `x` and `y` in the `self` object with them.

You may wonder where the `x` and `y` fields were declared—after all, we just start initializing them in the constructor. However, remember that Python doesn't have static type checking. When you initialize the field, it appears in the object. Note that if we had not written `self.` we would have been referring to *local* variables, even if they had names different from the parameters. Consider the following bad example:

```
class BadPoint:
    def __init__(self, x2, y2):
        x=x2
        y=y2
        pass
    pass
```

Here our constructor just makes two local variables, leaving the object with zero fields (technically, there are some built-in fields, such as `__class__`, but our bogus constructor had nothing to do with them).

Likewise, if we try to declare fields inside of the class, as we would in C++ or Java, we end up with the equivalent of *static* fields from those languages. For example,

```
class BadPoint:
    x=0
    y=0
    pass
```

creates two static (class) fields inside of `BadPoint`—so as with static variables you are familiar with, all instances of BadPoint will share the same value of `x` and `y`.

Returning to our `Point` class, we can create instances of it with `Point(x,y)` for whatever values of `x` and `y` we want:

```
p1=Point(3,4)
p2=Point(p1.x + 2, 42)
```

You can access the fields of the Point objects with the familiar dot operator (`p1.x`). However, remember that if you assign to a field that doesn't exist, it will come into existence with the assignment:

```
p1.z = 99
```

7.1 Methods

Now that we can create objects and modify/examine their fields, we would like to add some methods. We have seen `__init__`, which is the constructor for a class, and other methods are declared similarly: they look like function definitions inside the class, which take `self` as the first argument, and then any other arguments.

We'll start by defining two methods that every Python object should have: `__repr__` and `__str__`. (Note that while the methods we have seen so far have `__` on both sides of the name, that is not the common case for method name—it is used for certain special method names [2] These two methods specify how to convert the object to a string. The difference between them is that `__str__` gives a friendly representation for the outside world (*e.g.*, if you want to print the object), while `__repr__` gives a (possibly more detailed) representation of the object, which is printed when you evaluate an object in the Python interpreter. If you just evaluate your point now (*e.g.*, just evaluate `p1` after initializing it above), you get a pretty useless description. A first-pass change at fixing this description would be to write:

```
class Point:
    def __init__(self, x, y):
        self.x=x
        self.y=y
        pass
    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
    def __repr__(self):
        return "Point(" + repr(self.x) + ", " + repr(self.y) + ")"
    pass
```

[2] Names with two leading underscores and at most one trailing underscore, or with a single leading underscore have different special meaning.

Note that our `__repr__` method is a bit more explicit: it tells us that the object it is describing is a Point. Couldn't we infer that from output of `__str__`? That is, if we see `(5, 6)` should we not be able to tell that such a description is describing a Point? It could describe some other type with the same string representation. In fact, the tuple `(5, 6)` is *NOT* a Point object, but also has the same string representation. If we were debugging why something was behaving strangely, and the problem were that we were passing a tuple where a Point was expected, then having the same description would only further confuse use. However, explicitly printing the type name could help us clear up the issue.

Note that many Python programmers prefer to use Python's format strings (especially in 3.6 or later, where there is syntactic support for literals). Pre-3.6, we could write these two methods as:

```
    def __str__(self):
        return '({self.x}, {self.y})'.format(self=self)
    def __repr__(self):
        return '{self.__class__.__name__}({self.x!r}, {self.y!r})'.format(self=self)
```

And in Python 3.6, we could write

```
    def __str__(self):
        return f'({self.x}, {self.y})'
    def __repr__(self):
        return f'{self.__class__.__name__}({self.x!r}, {self.y!r})'
```

7.2 Inheritance

Python's classes support inheritance. When declaring the class, put the superclass(es) in parenthesis after the class name:

```
class Parent:
   # whatever declarations
class Child(Parent):
   # more declarations
```

Most things will work the way you expect with an inheritance relationship, especially if you are a Java programmer. If you are a C++ programmer, pretend that every method is declared `virtual`: they are all, always dynamically dispatched (after all, there are no static types, so how would they be statically dispatched?).

If you need to call a method from the parent class (*e.g.*., you are overriding the method, and want to perform the inherited functionality as well as new functionality), you can do this with `super().method(args)`. One important case of calling methods in the parent class arises in constructors: unlike C++ or Java, there is NO implicit call to the parent class constructor. Consider the following broken code:

```
class BrokenPoint3d(Point):
    def __init__(self, x, y, z):
        self.z = z
        pass
```

```
    def __str__(self):
        return '({self.x}, {self.y}, {self.z})'.format(self=self)
    def __repr__(self):
        return '{self.__class__.__name__}({self.x!r}, {self.y!r}, {self.z!r})'
                .format(self=self)

    pass
```

Here, the programmer attempts to inherit from Point to make a 3D point by adding a `z` field. However, there is no call to the parent class's `__init__`, so it does not happen. This oversight means that the 3d point object ends up only have a `z` field (and no `x` or `y` field)—which is only detected when the program attempts to read the `x` or `y` fields (if they are written first, they will come then exist, possibly making the problem even harder to find). Instead, we should have written:

```
class Point3d(Point):
    def __init__(self, x, y, z):
        super().__init__(x,y)
        self.z = z
        pass
```

If you want to inherit from multiple parent classes, list them, separated by commas:

```
class MIChild(Parent1, Parent2, Parent3):
  # declarations for this class here
```

We are not going to discuss multiple inheritance (you should know by now that it generally comes with a host of complexities)—if you want to use it, find a reputable source online and read about the details.

7.3 Access Restrictions

In C++ and/or Java, you are used to `private` and `protected` to enforce abstraction boundaries. Such things do not exist in Python (after all, these restrictions are enforced by the static type system of those languages). Instead, fields which start with a leading `_` are expected to be treated as if they were private by convention. However, the language does not enforce this convention.

7.4 Object Introspection

Python supports "object introspection"—the ability to examine the fields and methods of an object in other code. This is somewhat similar to Java's Reflection. We aren't going to go into detail on this beyond noting two useful things. The first is `dir`, which is a very useful function: it lists all the attributes (fields and methods) of the object passed to it.

The second thing we will note is the `help` function, which provides documentation about whatever is passed to it. Recall that we said at the start of this section that everything is an object. That means you can examine anything with these methods. Try out `help(3)` or `help(fancyfizzbang)` (assuming you defined it with the Docstring we discussed previously—you will see that), or `help([1,2,3])`.

8 Generators

Python has a special type of function called a *generator* which produces elements of a sequence, one-by-one. A generator `yield`s a value rather than `return`ing it. What exactly does this mean? It means that the function produces the value, allowing the consumer to use it, but remembers its local state so that it can produce the next value. This approach can be much more efficient than putting all the elements of a sequence into a list (or set) because it does not require all elements to be held in memory at once (or even to generate them all if whatever is iterating over the sequence only examines a few elements, then stops).

As an example, we could write a generator to produce the sequence of factorial numbers (0!, 1!, 2!, ...):

```
def factgen(n):
    temp=1
    for i in range(0,n):
        yield temp
        temp=temp*(i+1)
        pass
    pass
```

Notice how this yields the value rather than returning it, and there is computation which happens after the yield. If you try to evaluate a call to this function, *e.g.,* `factgen(6)`, you get a result like

```
<generator object factgen at 0x10f3c2ca8>
```

Note that this function has just made a generator object—it has not yet done any of the computation inside the factgen function. Instead, the computation will happen when we make use of the generator, typically by iterating over its elements:

```
for i in factgen(6):
  #...some loop body
```

How does the for loop iterate over the generator object? It calls the `.__next__()` method on the object to get the next value. If the execution of the generator function returns without yielding another value, then a StopIteration exception is raised, which is caught by the for loop implementation, causing it to leave the loop.

Now that you have the basics of generating function, let us look at a slightly more interesting example:

```
def strgen(n,items):
    if (n <=0):
        yield ""
    else:
        for base in strgen(n-1,items):
            for x in items:
                yield(base+x)
                pass
            pass
        pass
    pass
```

Take a moment and figure out what this function does. What sequence do you think `strgen(3,'01')` produces? If that seems to complicated, try `strgen(1,'01')` first. Try it out with `list(strgen(3,'01'))`. The list function will take the sequence and build a list from all of its elements. Did it give you what you expected?

Hopefully you figured out that this function generates the sequence of all possible combinations of length `n` of the elements of `items` (which can be any iterable type, as long as the elements are strings).

Note that this generator is much more interesting than factorial since it produces a very large sequence for even modest parameter sizes—namely, the sequence it produces has len(items)^n elements in it. We may not only save considerable memory by processing each element as we generate it (instead of storing all elements in a list), but also greatly improve locality: processing each element *immediately* after we produce it (rather than after producing the rest of the sequence). Also, we may save considerable time if whatever algorithm consumes the sequence decides that it does not need to examine all of them and can stop early.

9 Exercises 3

1. Write the generator `strgenShorter(n,items)` which generates the sequence of all combinations of elements of `items` of length less than `n`. For example, `strgenShorter(3,'ab')` would generate the sequence: '', 'a', 'b', 'aa', 'ab', 'ba', 'bb'. You should make use of `strgen` that from the example above.

10 Exceptions

Python has exceptions, which can be thrown (with `raise`) and can be handled with `try...except:`. The basic behavior of exception throwing and catching is pretty much what you would expect from C++ or Java (although if you are a C++ programmer, you may be disappointed to know that you cannot throw just any type: exceptions must inherit from BaseException—though any you define should inherit from its child class, Exception). There are some differences, which we will explore shortly, but first let us cover the basics a bit more concretely.

10.1 Raising exceptions

As an example of raising exceptions, we will start by writing a decorator that enforces a precondition on the value of a parameter passed to a function. Of course, you can raise exceptions in "regular" functions too—this is just a nice example of their use. In this example, you can see two uses of `raise`. One of which raises a TypeError, and one of which raises a ValueError:

```
def precond(argname,fn,mesg):
    def decorator(f):
        sig=inspect.signature(f)
        plist=list(sig.parameters.keys())
        if (not argname in plist):
            raise TypeError(f.__name__ + str(inspect.signature(f)) +
                            ' does not have ' + argname)
        @wraps(f)
```

```
        def wrapper(*args,**kwargs):
            v=sig.bind(*args, **kwargs).arguments[argname]
            if (not fn(v)):
                raise ValueError(mesg.format(val=v))
            return f(*args, **kwargs)
        return wrapper
    return decorator
```

Note that `inspect.signature` just introspects the function to find its signature,, and `sig.bind` maps the parameter names to their values—this lets us get the parameter value easily. We could then decorate a function to throw an exception if its parameter values are inappropriate:

```
@precond('n',
         lambda x: x>=0,
         'Argument must be greater than 0, but was {val}')
def fact(n):
  #...body of fact
```

10.2 Handling Exceptions

Handling exception is done with `try:...except:` (which is basically the same as try/catch). As with catch, you can (and should) specify the type(s) of exceptions to catch. If you write no type, then all exceptions will be caught (which may be a bad idea).

In general, exception handling code will look like this:

```
try:
   #code which might raise an exception
except ValueError:
   # code to handle value error
except (AttributeError, TypeError) as e:
   # code to handle either of these two types
   # note that e is bound to the exception
```

Here, we have two except blocks (we pick arbitrary exception types for the purposes of example—nothing special about them). The first catches `ValueError`s, but does not bind them to any name in the handler block. The second catches both `AttributeError`s and `TypeError`s. When it catches either one, the caught exception is bound to the variable `e` for the scope of the handler block.

10.3 Finally

Unlike C++, Python does not make use of RAII (although "with" which you will learn about shortly provides some of the same idea), so instead it has `finally`. If you are a Java programmer, the idea of `finally` is quite familiar: the code in a finally block always executes, whether or not an exception was raised or not. If an exception is raised, but is not handled by the `except` blocks, the code in the `finally` block will still execute and the exception will continue to propagate.

Note that (as with Java), finally may be used with or without except: blocks preceeding it. That is, you could have

```
try:
    #code
finally:
    #cleanup code
```

or you could have

```
try:
    #code
except SomeException:
    #handler code
except AnotherExn:
    #more handler code
finally:
    #cleanup code
```

If the try block is entered, then the finally block will ALWAYS execute (unless the program is killed by a fatal signal or something like that). Even things like returning out of the function, breaking out of a loop that contains the try block, or similar transfers of control will still result in the finally block executing before the function or loop is left. Even calling sys.exit (which exits the program) in the try block will still result in the finally block executing: sys.exit raises the SystemExit exception, which follows normal exception handling semantics.

10.4 Except/Else

Python also supports `else` between the `except` and `finally` blocks (or after `except`/before`finally` if only one of the two is present). Code in an `except` block is run if there were no exceptions in the `try` block, but before the `finally` is run. However, unlike code in the `try` block, the code in the `else` block will not have its exceptions caught by the `except` blocks. Here is an example of the general syntax:

```
try:
    #code
except SomeException:
    #handler code
except AnotherExn:
    #more handler code
else:
    #code to do if there are no exceptions
finally:
    #cleanup code
```

This construct is mostly useful if you have code which you want to execute at after what you wrote in the try block, but before what you wrote in the finally block—and this code (which goes in the else) might raise exceptions which you do not want to catch here.

11 Files, and with

In Python, you can open a file with the `open` function, which takes the name of the file to open as an argument. It also accepts the mode as a second argument, which default to 'r' for reading (you can use 'w' for writing, as well as some other possibilities—evaluate `help(open)` for details). The `open` function returns an object which can be used to manipulate the file (or raises an exception if it fails). You might be tempted to do something like:

```
f = open('somefile')
# do stuff with f
f.close()
```

While this approach works great when all goes well, it does not close `f` when an exception occurs (and you, of course, will write code which never leaks resources!). Instead, what you should do is use `with`, which provides a limited/manual RAII-like approach to resources:

```
with open('py-intro.tex') as f:
   #do stuff with f
```

`with` evaluates the expression immediately following it to obtain an object. It then records the `__exit__` function of this object, calls `__enter__()` on that object. If `__enter__` returns successfully, then `with` guarantees that it will call `__exit__()` on that object as control is transferred out of the `with` statement—whether normally (by reaching the end of the body), or by an exception being thrown. Note that `with` is not limited to files, you can use it with `Lock` as well, to get RAII-like mutex behavior. You could even write your own classes to use with `with`—just define `__enter__` and `__exit__` in them.

You can abbreviate nested `with`s with a comma-separated list of expression-as-name clauses in a single with:

```
with e1 as v1, e2 as v2, e3 as v3:
   # body
```

is short-hand for

```
with e1 as v1:
  with e2 as v2:
     with e3 as v3:
        # body
```

Now that you know how to *properly* open a file, you will want to be able to read or write from it. Reading from text files is most commonly done by iterating over them:

```
with open('myfile.txt') as f:
   for line in f:
     #something with each line
```

You can also use `f.read()` to read the *entire* file (even if it is huge...), or pass a size to `f.read` to read up to that many bytes from the file. You can also write to a file with `f.write(data)` (where data is the string you want to write to the file). There are a variety of other file related methods

in the object (seek, tell, etc): see `https://docs.python.org/3.5/tutorial/inputoutput.html` for reference.

We will also note that Python has support for reading/writing JSON—see the above linked documentation for details.

12 Exercises 4

1. Write the function `revLines` which takes 2 arguments: the first is the name of an input file, and the second is the name of an output file. It should read each line from the input file, reverse it, and write it to the output file.

2. Modify your `log` decorator such that if the function it decorates throws an exception, it will catch the exception, print a log message describing that outcome, and then re-throw the exception. This message should be another parameter (`ifexn`), and should support `$exn` to describe the exception. The default value for this parameter should be `"Calling $name($args) threw $exn"`.