Functions Script

Functions Script

Chapter 1

Quick refresher… look at a simple existing function

```
> python
>>> str = ”Hello world”
>>> length = len(str)
>>> print length
```

What have we done?

- Passed in parameter to function that returns length of string
- Received length of string, put in a variable
- Used the variable…

Define functions in Python using def

- Simple e.g. that just returns a string
- File: greet.py in a separate shell

```
def greet():
        return ‘Good evening, master’
```

- We need to call it
- Add… (then save)

```
result = greet()
print result
```

```
> python greet.py
```

- So, no problem

- Kind of useful as it is
- But far more useful is adding a parameter
- Say we want to greet a person by name

```
def greet(name):
        answer = ‘Hello, ‘ + name
        return answer

temp = ‘<get name>’
result = greet(temp)
print result
```

Far more useful!
- At basic level, follows same kind of pattern as len()
- Like elsewhere, note parameters are not typed
- But what is happening?
- [see ‘greet’ and stack frames in SC-FunctionsIntro.ppt, slides 5-9]

Exercise 1 – in pairs
- **Create a new function named exclaim() that takes one argument and is called by greet()**
- **exclaim(str) to adds an exclamation mark to a string and returns it before it is returned**

- But… use temp as a variable name in exclaim(str) to hold return value
- Check! Does the value of temp change in the global stack frame?

Chapter 2

So that’s basic functions, in a nutshell:
- Define your function
- Take in parameters
- Do something with those parameters
- Return a result

But we can ‘return’ at any time within a function
- In a new file e.g. sign.py:

```
def sign(num):
   if num > 0:
      return 1
   elif num == 0:
      return 0
   else:
      return -1
```

```
print sign(3)
print sign(0)
print sign(-9)
```

Essentially:

- If the number is positive (first if) return 1
- If the number is zero (second if) return 0
- If the number is negative (first if) return -1

But beware!

- Returning at anytime seems handy
- BUT… over-use can make functions hard to understand
- Reduce code readability
- Behaviour not always clear

So when to use return?

- Most programmers would agree that…
- One at the end to return the 'general' result
- But OK to have small number of early returns at very start of function (special cases)
- Keep it readable
- Python philosophy: code is read more often that it is written!

What happens if you don't use return??

- Comment out last two lines of sign() function, and rerun!

```
…
#   else:
#       return -1
```

- You get None
- All functions must return something. If not, they return None
- Generally not

Exercise 2 – rewrite sign() so that it only has one use of return

Exercise 3 - With the intro.py program I showed you yesterday, take highlighted blue code above and move it into a new function that takes a filename parameter and returns the number of lines. It should display the line count at the end as before.

Chapter 3

Beware Python types…

- Calling sign('Hello') doesn't make sense, but will run and return an answer!
- Try this… in e.g. double.py

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

print double(2)
```

- Run it, fair enough
- Now add…

```
print double('two')
```

- Eh? What happened?
- Turns out Python is quite happy to do this
- In Python, number * string is number occurrences of string concatenated
- Look out for this…

We've seen global vs current (or local) stack frames
- But consider…

```
def appender(a_string, a_list):
   a_string += 'turing'
   a_list.append('turing')

temp = 'alan'
list_val = ['alan']
appender(temp, list_val)

print temp
print list_val
```

Q: what's going to happen to temp and list_val?

In general:
- References to values are copied into parameters when function is called
- This creates aliases, strings are replaced
- But… when lists are passed into a function, it's passing in an alias for the list

In our case:
- Our global frame has two variables
- The call to appender places a new frame with aliases for those variables on top of the stack
- a_string += 'turing' creates a new string and overwrites the value of a_string
- BUT… a_list.append('turing') actually modifies what a_list is pointing to
- When function returns, 'alanturing' new string is lost
- And the change to list is kept
- Useful! Follows very common pattern of using big data

Assume we're writing a large piece of Python…
Q: when should you start thinking about decomposing it into functions?

- No hard and fast rule
- When you see a 200 LOC function, you see a breeding ground for bugs
    - Some exceptions – large switch/case statements
- Want functions to fit into short-term memory – 7/8 things at once
- For functions, aim for this balance of function length
- Gerald (Jerry) Weinberg's "Psychology of Computer Programming" (1971)
    - Still in print – Silver Edition
    - Multiple studies have shown that programmer comprehension essentially limited to what the programmer can see at any given instant
    - If he has to scroll, or turn a page, his comprehension drops significantly
    - Backed up by others (Robert Martin)
- More controversy recently (Steve McConnell)
    - 100-200 lines - decades of evidence says such routines no more error prone
- Good rule of thumb is complexity of function to dictate function size
- Make them easy to understand
- H.Glaser's example of fitting program into cache, a lot slower when something arbitrary was added, didn't know why, diagnosis was difficult. Program no longer fitted into cache!
- Always thought that this was a nice simile for memory. Think of the CPU as your brain, and its cache as short term memory! Or that of anyone reading your code!

Good practice is to write the high-level code first

- Make use of functions you have yet to write
- Then go back and write the functions
- Get high-level flow right. Delving into functions disrupts flow
- Breadth-first vs depth-first use of functions
- Depth-first: easy to do, risk getting lost in the depths, thinking what else you may use function for later, coding accordingly
- Breadth-first: much better idea of what the functions need to do

Exercise 4(a)

- **Assume we have a separate Python library functions.py that contains our greet, sign and double functions**
- **Write a new Python program e.g. prog.py that**
    - **Greets you by name**
    - **Loops through the numbers 3, 0 and -9 and runs sign and double on them, printing each result**
    - **Do not use, rewrite or run the greet, sign or double functions until you have finished prog.py!**

Exercise 4(b)

- **Take greet, sign and double functions and put them in a separate functions.py file, and import them. Change prog.py to use them**
- **Now you can run your prog.py!**

- **Can also use ‘from functions use fun_a, fun_b, fun_c’ instead of import – don’t need to use ‘functions’ prefix. Change prog.py to use this method of importing functions**

```
from functions import greet, sign, double

print greet('Name')

arr = [3, 0, -9]
for num in arr:
   print 'Number', num
   print 'Sign', sign(num)
   print 'Double', double(num)
```

Chapter 4

We’ve seen stack frames, and how they handle variables i.e. data

- But aren’t functions at the end of the day just data as well?
- Could we pass them around in variables and use them?
- Yes! Idea been around a long time – functional programming
- Let’s try this out in our prog.py

```
from functions import greet, sign, double

print greet('Name')

s = sign
d = double

arr = [3, 0, -9]
for num in arr:
   print 'Number', num
   print 'Sign', s(num)
   print 'Double', d(num)
```

So far, so useless

- Why would we want to do this??
- Functions that use functions!
- Let’s look at our prog.py
- Copy it to another file e.g. prog-arrays.py

Let’s assume we need to hold separate arrays for each set of answers for array arr

- We could do it this way…

```
from functions import greet, sign, double
```

```
print greet('Name')

arr = [3, 0, -9]

signed = []
for num in arr:
   temp = sign(num)
   signed.append(temp)

doubled = []
for num in arr:
   temp = double(num)
   doubled.append(temp)

print 'started with', arr
print 'sign', signed
print 'doubled', doubled
```

But – we have code duplication in the loops

- With a larger set of functions we could introduce errors
- (through copy and paste)
- Unnecessary code duplication = bad
- What if the style of the loop in general needs to change?
- Maintenance problem!

Instead, using functional programming…

```
from functions import greet, sign, double

def do_for_each(func, values):
   result = []
   for v in values:
      temp = func(v)
      result.append(temp)
   return result

print greet('Name')

arr = [3, 0, -9]

signed = do_for_each(sign, arr)
doubled = do_for_each(double, arr)

print 'started with', arr
print 'sign', signed
print 'doubled', doubled
```

- Reduced testing with this kind of logic – test do_for_each, not a separate list of loops
- This method relates back to science really well - e.g. max(f, x0, x1). f could be sine, cosine, etc. Normal in maths of functions doing stuff to other functions
- But there's always a tradeoff - more difficult for compiler to optimise
- This is basis of mapreduce used at Google; fits a lot of scientific models

Exercise 5 – add in another mathematical function of your choice to functions.py and use it in the same fashion on the array arr