Last week

- Last week: We learned how to do simple procedural programming Python.

```
for i in range(0,10):
        print i
```

```
def fibonacci( n ):
    a = 1
    b = 1
    for i in range(1,n):
        a,b=b,a+b
    return a
```

- This week: flow of control.

Errors

- You can create an error using `raise`
- You can recover from an error using `try` and `except`
- You can execute some code whether an error occurs or not using `finally`

```
def debit_account( account, amount ):
    if amount>account:
        raise ValueError("You have insufficient funds")
    return account-amount

account = 100; amount = 20
try:
    account = debit_account(account, amount)
    print("Account debitted")
except ValueError:
    print("Failed to debit account")
finally:
    print("An attempt was made to debit from the account")
```

Using Errors

- Never just print out a message in response to an error.
- Handling errors is for expert programmers. Don't bother yourself.
- There are different types of error, e.g. `ValueError` or simple `Exception`. For "normal" errors it doesn't matter too much what exception type you use as no-one will be able to handle the error anyway
- Make heavy use of assert
- Finally clauses are designed for situations like database connections or files that you must close when you have finished with them

Generators

Generators provide a cool way of returning a sequence of values without needing to hold the entire sequence of values in memory

```
def is_fibonacci_2( x ):
    for fib in generate_fibonacci():
        if fib==x:
            return True
        if fib>x:
            return False
```

```
def generate_fibonacci():
    a = 1
    b = 1
    while True:
        yield a
        a,b=b,a+b
```

Working with iterators

Working with iterators. You can use `iter` to turn a generator into something you can iterate through with next()

```
def test_generate_fibonacci():
    all_fib = iter(mymath.generate_fibonacci())
    first_six = []
    for i in range(0,6):
        first_six.append( next(all_fib))
    nose.tools.assert_equals(first_six, [1, 1, 2, 3, 5, 8])
```

Ending iteration

```
def my_range( start, end ):
    i = start;
    while i<end:
        yield i
        i+=1
    raise StopIteration()


# Usage example
r = iter( my_range(1,6))
try:
    while (True):
        print( str(next(r)))
except StopIteration:
    pass
```

The pass statement does nothing. Use it when a statement is required by Python, but you don't want to do anything.

The list function

For finite iterators, you can use the list function

```
print( range(0,6))
print( list(range(0,6)) )
```

Lists use up more memory than iterators. Clearly

```
list( mymath.generate_fibonacci())
```

is never going to work. Iterators are great for, say, reading lines of a large file.

Break

```python
def is_prime_1(n):
    for x in range(2, n):
        if n % x == 0:
            return False
    return True
```

An alternative

```python
def is_prime_2(n):
    x = 2
    while True:
        if x>=n:
            return True
        if n % x == 0:
            break
        x+=1
    return False
```

Break and else

Another alternative

```
def is_prime_3(n):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        return True
    return False
```

Continue

The continue statement.

```
def find_factors(n):
    factors = []
    for x in range(1, n+1):
        if not( n % x == 0):
            continue
        factors.append(x)
    return factors
```

Try and else

You can use `else` with `try` too

```
try:
    debit_account(account, amount )
except ValueError:
    print('Could not debit account')
    raise
else:
    print('Debitted account')
```

`raise` without any arguments re-raises an exception. If you insist on catching exceptions and printing them out, you should normally re-raise them.

Working with files

- You must call `open` to start writing to a file. This opens a "stream"
- You must call `close` when you have finished with the file.

```
f = open('testfile.txt','w')
f.write('This is a line\n')
f.write('This is another line\n')
f.close()
```

Gotcha

- Errors can occur at any time. They typically happen when you least expect them

```
f = open('testfile.txt','w')
i = iter(range(1,6))
f.write(str(next(i)))
f.close()
```

Fix it with finally

- Fix it with finally (or even better...)

```
f = open('testfile.txt','w')
try:
    i = iter(range(1,6))
    f.write(str(next(i)))
finally:
    f.close()
```

Fix it with with

- Fix it with with (stet)

```
with open('testfile.txt','w') as f:
    i = iter(range(1,6))
    f.write(str(next(i)))
```

- With works with data objects that have a `close` function

Tips

- `raise` is good, `assert` is even better
- `yield` is good
- I'd avoid `except` except with iterators
- `break`, `continue` aren't so useful
- `else` with `for` isn't so useful
- `else` with `try` isn't so useful
- `finally` with `try` is useful, but `with` is better.

Bible Study

Download the Bible from `http://www.gutenberg.org/cache/epub/10/pg10.txt` and save it as 'bible.txt'.

```
line_count = 0
with open('bible.txt','r') as f:
    for line in f:
        line_count = line_count+1;
print('There are '+str(line_count)+' lines in the Bible');
```

Exercises

- Use `s.split` to find all the words in a string s. How many words are there in the Bible?
- What is the first line in the Bible containing the word Beelzebub? Use `in` to see if a string contains another string.
- How often does the word God appear in the Bible? Use lower to check how often the word God or god appears.
- Write a function `bible_words()` which uses `yield` to create an iterator through all the words in the Bible. Why might it not be a good idea to create a list of all the words in the Bible?
- Use `len` to find the length of a string. What is the longest word in the Bible? To answer this, write a function `longest_item` which finds the longest item coming from any iterator.
- What does your function `longest_item` do if it is given an empty iterable? What should it do?

Finding God

The best way to do this is using a 'regular expression' from the package re. We compile a pattern and can then check if our pattern matches an string.

```
import re

def countGodRe():
    pattern = re.compile('\W*God\W*')
    god_count = 0
    with open("bible.txt", "r") as f:
        for line in f:
            for word in line.split():
                if pattern.match( word ):
                    god_count=god_count+1
    print ('The word God appears '+str(god_count)+' times.')
```

Regular expressions

- `.` means any character, `*` means zero or more times so `.* Station` matches "Paddington Station" and "Charing Cross Station"
- `\w` means any "word character", i.e. a letter or a numbers. So `\w* Station` matches "Paddington Station" but not "Charing Cross Station"
- `\W` means any "non=word character", i.e. anything other than a letter or a number such as a punctuation mark. `\W*God\W*` matches "God" and "God," and "'God!'" but not "Godless"

Regular expressions are a powerful tool for manipulating text that are well worth learning if you ever need to perform some repetitive search and replace tasks.

Objects and Classes

- `f` is a file object with useful functions like `write`, `readline`, `next`
- `line` is a string object with useful functions like `split` and `lower`
- `pattern` is a regular expression pattern object with usefull functions like `match`
- Most of the time you use a `.` to access a function. Python also provides some shorthands, for example the command `'god' in line` is equivalent to `line.__contains__('god')`.
- We will learn how to write our own classes of object next week.

Optional and named parameters

```
def print_number( number, base=10, units='', currency=''):
assert (base<=10)
print(currency, end='')
digits = []
if number==0:
digits.append('0')
while number>0:
digit = number % base
digits.append(str(digit))
number = number//base
for digit in reversed(digits):
print(digit, end='')
print(units, end='')
print() # inserts newline
```

e.g. `mymath.print_number(100,units='kg',base=8)`

Optional and named parameters

```
def print_number( number, base=10, units='', currency=''):
    assert (base<=10)
    print(currency, end='')
    digits = []
    if number==0:
        digits.append('0')
    while number>0:
        digit = number % base
        digits.append(str(digit))
        number = number//base
    for digit in reversed(digits):
        print(digit, end='')
    print(units, end='')
    print() # inserts newline
```

e.g. `mymath.print_number(100,units='kg',base=8)`

Multiple argument lists

Occasionally it is useful to write functions that take multiple arguments

```python
def concatenate_strings(*strings, sep='', include_last=False):
    ret = ""
    for i in range(0,len(strings)):
        s = strings[i]
        ret +=str(s)
        if i<len(strings)-1:
            ret +=sep
        elif include_last:
            ret +=sep
    return ret

print(concatenate_strings(1,2,3,4,sep='+'))
```

Unpacking argument lists

Multiple arguments are very convenient if the user hasn't yet created a list and saves them the bother. However, it can be annoying if you already have a list.

```
four_numbers = [1,2,3,4]
concatenate_strings(four_numbers,sep='+')
```

The code above doesn't do what we want. Note the $*$ in the code below.

```
four_numbers = [1,2,3,4]
concatenate_strings(*four_numbers,sep='+')
```

List comprehensions

```
squares = [x**2 for x in range(10)]

n = 10
triads = [(a**2-b**2, 2*a*b,a**2+b**2) \
          for a in range(1,n) for b in range(1,n) if b<a]
```

This is pretty slick

Map and lambda functions

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

l = range(1,100,2)
sl = list(map( square_it, l ))
print( sl )

sl = list(map( lambda x: x**2, l ))
print( sl )
```

You can apply `map` to anything you can iterate over, i.e. anything you can use in `for` statements.

Comparison with Mathematica

- List comprehensions are very like the Table command
- Mathematica has an `If` function and a `For` function

Sets

- Use `{}` for sets
- Set comprehensions work as you would expect
- The function `set` works
- The `in` statement works as you would hope to see if an element is in a set

```
squares = {x**2 for x in range(-10,11)}
assert( len(squares)==11 )
assert( 16 in squares )
assert( 17 not in squares )
```

Dictionaries

```
dict = { 'horse':'hairy quadruped',
         'bird':'feathery biped',
         'man':'biped without feathers' }
for key,value in dict.items():
    print('A {} is a {}'.format(key,value))

dict['fish']='legless thing, but not a plant'

for key in dict:
    print('A {} is a {}'.format(key,dict[key]))
```

Dictionaries allow you to look up and store values quickly

Global variables

```
bill = 0

def expensiveFunction():
    global bill
    bill +=100

def printBill():
    global bill
    print('Your total bill is {}'.format(bill))

for i in range(0,100):
    expensiveFunction()
printBill()
```

Summary

- We have seen the procedural programming style today
    - The functions we have written are the "procedures"
    - There are a large number of "flow of control" statements
    - You know exactly what tasks the computer performs and in what order
    - The values of variables changes often, in strict functional programming the values of a variable never change
- We have seen that Python has good support for unit testing. Use it.

Exercises - Page 1

- Implement the sieve of Eratosthenes in Python. Write as few lines of code as possible if you want to show off.
- Write a unit test for the above.
- Write a recursive function of `fibonacci` that uses a global dictionary to be efficient
- Write a function `concatentate` that takes two iterable objects and returns a single iterable object that joins them together.
- Enhance `concatentate` so that it takes an arbitrary number of iterable objects and returns a single iterable object that joins them together. `itertools.chain` is the built in way to do this.

Exercises - Page 2

- Create a table of the frequency of use of each letter in the Bible.
- Write a function `merge` that takes two iterable objects that each return numbers in order and creates a single ordered iterator. Note that I found this pretty challenging myself.
- Implement a merge sort by recursion. (Of course, Python has the `sorted` function you want already. My merge sort implementation is 100 times slower than this)