

# Functions

Week 2, day 2

# Functions

- Saves repetition  
(creative laziness)
- Helps  
modularisation &  
readability
- *Returns, args, docstrings, name,  
lambdas, recursions*

```
def hamming (s1, s2):  
    theDist = 0  
    ln = len (s1)  
    for i in range (ln):  
        if s1[i] != s2[i]:  
            theDist += 1  
    return theDist
```

# Multiple returns

- Can return multiple values
- Implicit tuple
- “Pythonic”

```
a, b = findSpliceSite()
```

```
[...]
```

```
def getSpliceSite ():  
    start = seq.find  
        ('ACT')  
    stop = seq.find  
        ('GGT', start)  
    return start, stop
```

# “Docstrings”

- First string
- Provides documentation for `??`, `doc()` etc.
- Not just functions

```
def getSpliceSite ():  
    "Returns the beginning  
    & end of the probable  
    splice site"  
    start = seq.find  
        ('ACT')  
    stop = seq.find  
        ('GGT', start)  
    return start, stop
```

# Arguments are references

- A pointer to the actual value not the value itself

```
myLst = [1, 2, 3, 4]

def addToLst (x, lst):
    lst.append (x)

# myLst is ...?
```

# Keyword arguments

- Can name args when passing
- Don't need to keep order
- Readability

```
def findSplice (start_str,
                stop_str):
    start = seq.find (start_str)
    stop = seq.find (stop_str, start)
    return start, stop

# these 3 calls are identical
a, b = findSplice('AGT', 'CCT')

a, b = findSplice(start_str='AGT',
                  stop_str='CCT')

a, b = findSplice(stop_str='CCT',
                  start_str='AGT')
```

# Default arguments

- Don't need to use all arguments
- Just name the ones used
- Provide useful default behaviour

```
def findSplice (start_str='AGT',
                stop_str='CCT'):
    start = seq.find (start_str)
    stop = seq.find (stop_str, start)
    return start, stop

# these 3 calls are identical
a, b = findSplice()

a, b = findSplice(start_str='AGT')

a, b = findSplice(stop_str='CCT')
```

# Case study: string find

```
# write a fxn like string find:  
# find (str, substr [,start] [, stop])
```

# \*args : All args

- collects all remaining parameters in list
- error if unhandled

```
# with this call ...
a, b = findSplice('AGT', 'CCT')

# and these definitions
def findSplice (*args):
    [...]

def findSplice (start, *args):
    [...]

def findSplice (start, stop, *args):
    [...]
```

# **\*\*kwargs : All keyword args**

- collects all remaining keyword parameters in dict
- error if unhandled

```
# with this call ...
a, b = findSplice(start='AGT',
                  stop='CCT')

# and these definitions
def findSplice (**kwargs):
    [...]

def findSplice (start='CC', **kwargs):
    [...]

def findSplice (start='CC', stop='GG',
                **kwargs):
    [...]
```

# \*args & \*\*kwargs together

- actual parameter names are args & kwargs

```
def findSplice (*args, **kwargs):  
    for item in args:  
        print item  
    for k, v in kwargs.items():  
        print k, v
```

# Case study: print

```
# write a fxn like print:  
# print (a, b, c, d, e ...)
```

# lambda: anonymous functions

- Construct short functions on the fly
- Can treat as objects ...

```
>>> x = lambda s,a,b : s[a:b]
>>> st = "abcdefghij"
>>> x (st, 1, 4)
"bcd"
```

# Functions are objects

- Can pass around like variables
- Assign, hand to functions ...
- Not methods ...

```
import re  
  
mycomp = re.compile  
  
myRegex = mycomp (  
    r'\d*\.\d')
```

# Methods are functions

- Associated with an object
- Attribute of an object
- Also objects but ...

```
>>> st = "abcdefghi"
>>> len (st)
9
>>> st.find ('bcd')
1
>>> x = st.find
>> x ('bcd')
1
```

# Why treat functions as objects?

- Functional programming
- Efficiency
- Function is “saved” to act later

# Recursion

- A function that calls itself
- Decomposable problems
- Efficient expression
- “Maximum recursion depth exceeded”

```
def power (b, p):  
    """  
    Return b to the power of p.  
    """  
    if (p == 1):  
        return b  
    else:  
        return b * power (b, p-1)
```

# Case study: binary search

```
# search a sorted list for an entry
```

# Scope

- Global scope & local scope
- Implied scope

```
>>> x = 5

>>> def changex():
    x = 0
    print x

>>> changex()

>>> print x

>>> def readx():
    y = x
    print y

>>> readx()
```

# Global

- Forces variable to be interpreted as global

```
>>> x = 5

>>> def changex():
        global x
        x = 0
        print x

>>> changex()
0
>>> print x
0
```