Genome Sciences 373
Genome Informatics

Quiz Section 5

April 28, 2015

Functions in Python: a brief overview

Functions are:

reusable pieces of code, that take zero or more **arguments**, perform some **actions**, and **return** one or more values

Functions in Python: a brief overview

Functions are:

reusable pieces of code, that take zero or more **arguments**, perform some **actions**, and **return** one or more values

conceptually

function "**sum**"
takes arguments a, b
adds a and b
returns sum

Functions in Python: a brief overview

Functions are:

reusable pieces of code, that
take zero or more **arguments**,
perform some **actions**, and
return one or more values

conceptually

function "**sum**"
takes arguments a, b
adds a and b
returns sum

in python...

```
def sum(a, b):
    total = a + b
    return total

# later in the program
my_sum = sum(2, 5)
# my_sum is now 7
```

Functions in Python: a brief overview

Functions are:

reusable pieces of code, that take zero or more **arguments**, perform some **actions**, and **return** one or more values

stuff that happens in here is invisible outside of the function

in python...

```
def sum(a, b):
    total = a + b
    return total

# later in the program
my_sum = sum(2, 5)
print total   # this won't work!
```

```
def jc(seq1, seq2):
        # find the length of the alignment
        seqlength = len(seq1)

        # counters
        informative_pos = 0
        mismatch_pos = 0

        # progress through the sequence
        for index in range(seqlength):
                # ignore gaps
                if seq1[index] == "-" or seq2[index] == "-":
                        continue
                # look for mismatches
                if seq1[index] != seq2[index]:
                        mismatch_pos += 1
                # increment the counter of informative positions
                informative_pos += 1

        # find the raw distance: number of mismatches divided by the number
        # of informative positions
        raw_distance = float(mismatch_pos) / float(informative_pos)

        # calculate the Jukes-Cantor distance from the raw distance
        jc_distance = -0.75 * math.log(1.0 - (4.0/3.0 * raw_distance))
        return jc_distance
```

function to find Jukes-Cantor distance

Functions in Python: returning stuff

Functions can return more than one value:

```
def sum_and_product(myList):
   num_sum = 0
   num_product = 1
   for num in myList:
      num_sum += num
      num_product *= num

   return [num_sum, num_product]

>>> x = sum_and_product([1, 2, 4])
>>> print x
[7, 8]
```

In-class example:

Write a function to calculate the factorial of an integer

Inputs: *integer >= 0*

Outputs: *integer > 0*

The intuitive approach

```
def factorial(x):
  my_total = 1
  while x > 1:
    my_total = my_total * x
    x -= 1
  return my_total

print factorial(10)
# prints 3628800
```

The recursion approach

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)

# (will print 3628800)
```

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

step 1: 10 * fact (10 – 1)

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

```
step 1: 10 * fact (10 – 1)
step 2: 10 * 9 * fact(9 – 1)
```

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

```
step 1: 10 * fact (10 – 1)
step 2: 10 * 9 * fact(9 – 1)
step 3: 10 * 9 * 8 * fact(8 – 1)
```

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

```
step 1: 10 * fact (10 – 1)
step 2: 10 * 9 * fact(9 – 1)
step 3: 10 * 9 * 8 * fact(8 – 1)
step 4: 10 * 9 * 8 * 7 * fact(7 – 1)
[…]
step 8: 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * fact(3-1)
```

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

```
step 1: 10 * fact (10 – 1)
step 2: 10 * 9 * fact(9 – 1)
step 3: 10 * 9 * 8 * fact(8 – 1)
step 4: 10 * 9 * 8 * 7 * fact(7 – 1)
[…]
step 8: 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * fact(3-1)
```

```
def fact(x):
    if x == 2:
        return x
    else:
        return x * fact(x - 1)

print fact(10)
```

```
step 1: 10 * fact (10 – 1)
step 2: 10 * 9 * fact(9 – 1)
step 3: 10 * 9 * 8 * fact(8 – 1)
step 4: 10 * 9 * 8 * 7 * fact(7 – 1)
[…]
step 8: 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * fact(3-1)
step 9: 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 # done!
```

A more complete version…

```
def fact(x):
    if x < 0:
        # write error and exit
        sys.stderr.write(…)
    if x == 0 or x == 1:
        return 1
    if x == 2:
        return x
    else:
        return x * fact(x - 1)
```

Functions in Python: required arguments

```
def less_than(myList, num):
   new_list = []
   for x in myList:
      if x < num:
         new_list.append(x)
   return new_list

>>> my_list = [1, 2, 3]
>>> m = less_than(my_list)
>>> print m
TypeError: less_than() takes exactly
2 arguments (1 given)
```

Functions in Python: required arguments

```
def less_than(myList, num):
    new_list = []
    for x in myList:
        if x < num:
            new_list.append(x)
    return new_list

>>> my_list = [1, 2, 3]
>>> m = less_than(my_list, 3)
>>> print m
[1, 2]
```

Functions in Python: default arguments

```
def less_than(myList, num=4):
    new_list = []
    for x in myList:
        if x < num:
            new_list.append(x)
    return new_list

>>> my_list = [1, 2, 3]
>>> m = less_than(my_list)
>>> print m
[1, 2, 3]
```

Functions in Python: default arguments

```
def less_than(myList, num=4):
    new_list = []
    for x in myList:
        if x < num:
            new_list.append(x)
    return new_list

>>> my_list = [1, 2, 3]
>>> m = less_than(my_list, 2)
>>> print m
[1]
```

You can override default arguments