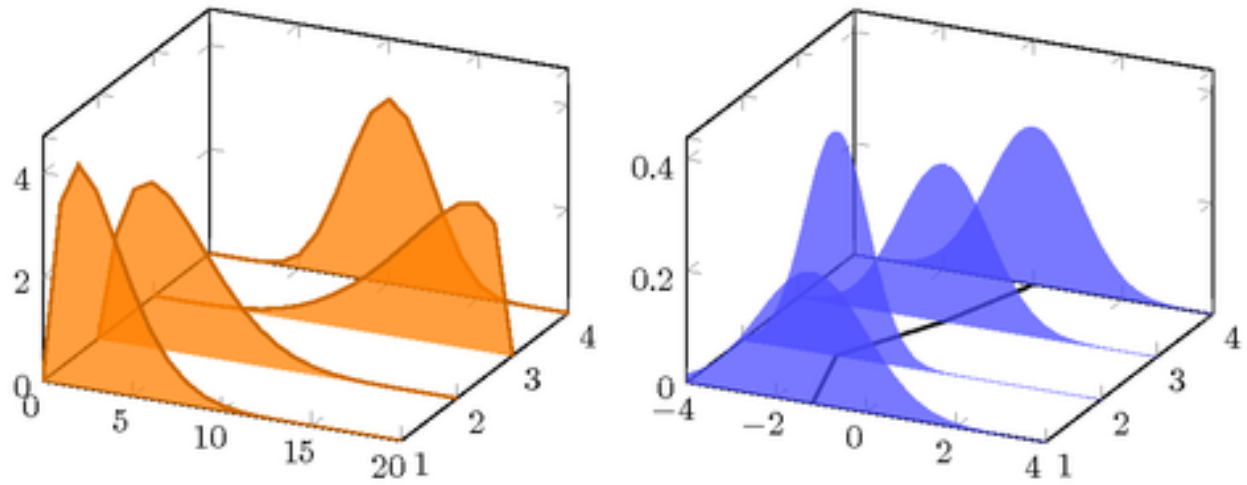
Functions

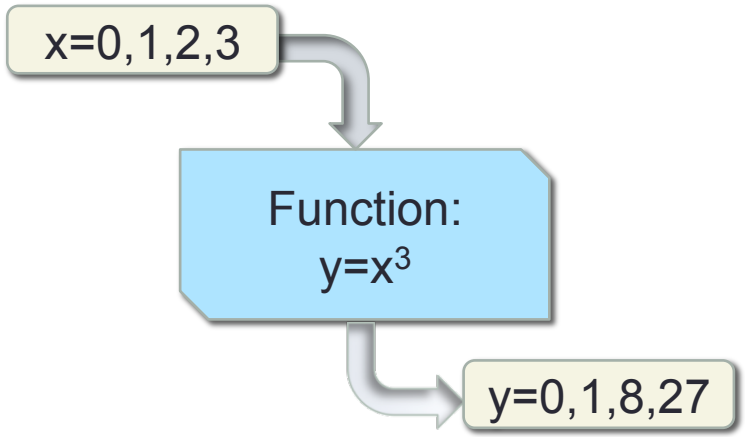
Python for Genomic Data Science



What Are Functions?

A *function* is a part of a program. It takes a list of argument values, performs a computation with those values, and returns a single result.

Python gives you many *built-in* functions like `print()`, `len()` etc. but you can also create your own functions. These functions are called *user-defined functions.*



Why Write Functions?

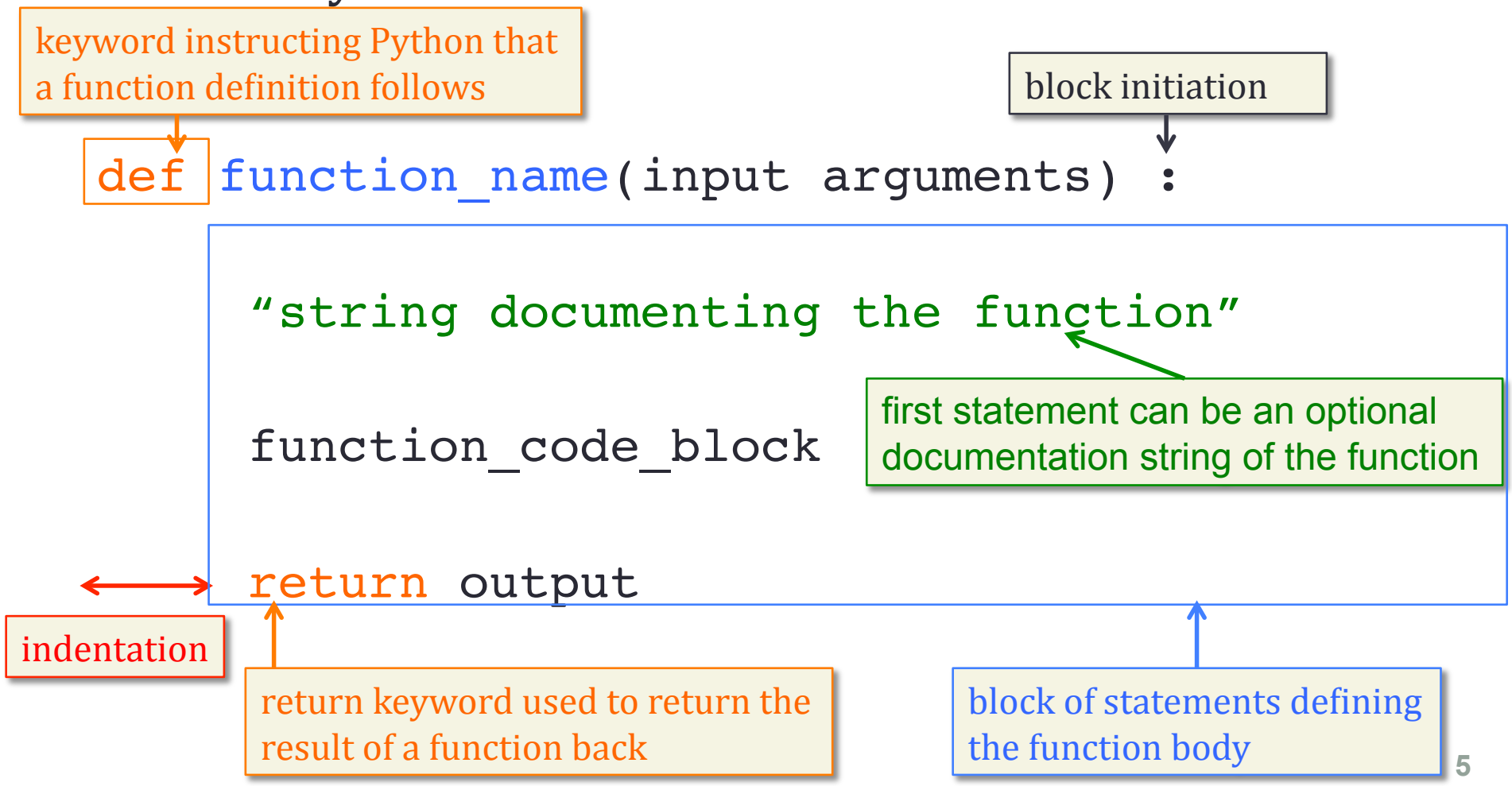
1. *Reusability*: They allow us to reuse code instead of rewriting it. Once a function is defined, it can be used over and over again. You can invoke the same function many times in your program, which saves you work. You can also reuse function written in different programs.
2. *Abstraction*: They allow us to conceive our program as a sequence of sub-steps. You know what each sub-step in your program does, but you don't necessarily need to know how it does it – i.e. like when you use functions written by other people.

Some Useful DNA Sequence Functions

1. A function that computes the GC percentage of a DNA sequence
2. A function to check if a DNA sequence has an in frame stop codon.
3. A function to reverse complement a DNA sequence.

Defining Functions

General syntax:



A Function To Compute The GC Percentage Of A DNA Sequence

Input : DNA sequence
Output : GC percentage

```
>>> def gc(dna) :
...     "this function computes the GC percentage of a dna sequence"
```

A Function To Compute The GC Percentage Of A DNA Sequence

Input : DNA sequence
Output : GC percentage

```
>>> def gc(dna) :
...     "this function computes the GC percentage of a dna sequence"
...     nbases=dna.count('n')+dna.count('N')
...     gcpercent=float(dna.count('c')+dna.count('C')+dna.count('g')
+dna.count('G'))*100.0/(len(dna)-nbases)
...     return gcpercent
```

A Function To Compute The GC Percentage Of A DNA Sequence

Input : DNA sequence
Output : GC percentage

```
>>> def gc(dna) :
...     "this function computes the GC percentage of a dna sequence"
...     nbases=dna.count('n')+dna.count('N')
...     gcpercent=float(dna.count('c')+dna.count('C')+dna.count('g')
+dna.count('G'))*100.0/(len(dna)-nbases)
...     return gcpercent

>>> gc('AAAGTNNAGTCC')
0.4

>>> help(gc)
Help on function gc in module __main__:

gc(dna)
    this function computes the GC percentage of a dna sequence
```

Scope Of Variable Declaration

If you declare a variable inside a function it will only exist in that function:

Scope Of Variable Declaration

If you declare a variable inside a function it will only exist in that function:

```
>>> def gc(dna) :
...     "this function computes the GC percentage of a
dna sequence"
...     nbases=dna.count('n')+dna.count('N')
...     gcpercent=float(dna.count('c')+dna.count('C')
+dna.count('g')+dna.count('G'))/(len(dna)-nbases)
...     return gcpercent

>>> print(nbases)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    print(nbases)
NameError: name 'nbases' is not defined
```

Boolean Functions

Boolean functions are *functions* that return a True or False value. They can be used in conditionals such as `if` or `while` statements whenever a condition is too complex.

Problem. Write a program that checks if a given DNA sequence contains an in-frame stop codon.

dna_has_stop.py

```
#!/usr/bin/python

# define has_stop_codon function here

dna=    input("Enter a DNA sequence, please:")

if(has_stop_codon(dna)) :
    print("Input sequence has an in frame stop codon.")
else :
    print("Input sequence has no in frame stop codons.")
```

A Function To Check For In-Frame Stop Codons

```
def has_stop_codon(dna) :
      "This function checks if given dna
sequence has in frame stop codons."
      stop_codon_found=False
      stop_codons=['tga','tag','taa']
      for i in range(0,len(dna),3) :
            codon=dna[i:i+3].lower()
            if codon in stop_codons :
                  stop_codon_found=True
                  break
      return stop_codon_found
```

Defining Function Default Values

Suppose the has_stop_codon function also accepts a frame argument (equal to 0,1, or 2) which specifies in what frame we want to look for stop codons.

```
def has_stop_codon(dna,frame) :
        "This function checks if given dna sequence
has in frame stop codons."
        stop_codon_found=False
        stop_codons=['tga','tag','taa']
        for i in range(frame,len(dna),3) :
                codon=dna[i:i+3].lower()
                if codon in stop_codons :
                        stop_codon_found=True
                        break
        return stop_codon_found
```

Defining Function Default Values (cont'd)

```
>>> dna="atgagcggccggct"
>>> has_stop_codon(dna,0)
False
>>> has_stop_codon(dna,1)
True
```

We would like the `has_stop_codon()` function to check for in-frame stop codons by default, i.e. if we call the function like this:

```
>>> has_stop_codon(dna)
```

to assume that `frame` is 0.

Defining Function Default Values (cont'd)

To use the 0 value as the default value for the `frame` parameter, the `has_stop_codon` function can be redefined as follows:

```
def has_stop_codon(dna,frame=0) :
        "This function checks if given dna sequence has in frame stop codons."
        stop_codon_found=False
        stop_codons=['tga','tag','taa']
        for i in range(frame,len(dna),3) :
                codon=dna[i:i+3].lower()
                if codon in stop_codons :
                        stop_codon_found=True
                        break
        return stop_codon_found
```

```
>>> dna="aaatgagcggccggct"
>>> has_stop_codon(dna)
True
>>> has_stop_codon(dna,1)
False
```

Passing Function Arguments

Passing arguments *by position:*

```
>>> seq='tgggcctaggtaac'
>>> has_stop_codon(seq,1)
True
```

It is also possible to pass arguments *by name*:

```
>>> has_stop_codon(frame=0,dna=seq)
True
```

the order of arguments is no longer required

You can mix either style, but named argument must be put after positioned arguments:

```
>>> has_stop_codon(seq,frame=2)
False
```

A More Complex Function Example

Write a function to reverse complement a DNA sequence.

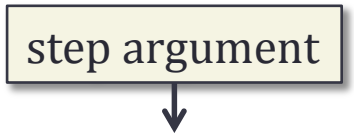
```
>>> def reversecomplement(seq):
      """Return the reverse complement of the dna
string."""
      seq = reverse_string(seq)
      seq = complement(seq)
      return seq

>>> reversecomplement('CCGGAAGAGCTTACTTAG')
'CTAAGTAAGCTCTTCCGG'
```

Reversing A String

Regular slices

```
>>> dna="AGTGTGGGGCG"
>>> dna[0:3]
'AGT'
```

Extended slices




```
>>> dna[0:3:2]
'AT'
>>> dna[::-1]
'GCGGGGTGTGA'
```

```
>>> def reverse_string(seq):
        return seq[::-1]

>>> reverse_string(dna)
'GCGGGGTGTGA'
```

A Function To Complement A DNA Sequence

```
def complement(dna):
    """Return the complementary sequence
string."""
    basecomplement = {'A':'T', 'C':'G',
'G':'C', 'T':'A', 'N':'N', 'a':'t', 'c':'g',
'g':'c', 't':'a', 'n':'n'}
    letters = list(dna)
    letters = [basecomplement[base] for
base in letters]
    return ''.join(letters)
```

List Comprehensions

- *List comprehensions* in Python provide a concise way to create lists.
- Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence, or to create a subsequence of those elements that satisfy a certain condition.

Syntax

```
new_list = [operation(i) for i in old_list if filter(i)]
```

or

```
new_list = []
for i in old_list:
    if filter(i):
        new_list.append(operation(i))
```

How To Complement Each Letter In A DNA Sequence

```
>>> dna
'AGTGTGGGGCG'
>>> basecomplement = {'A':'T', 'C':'G', 'G':'C',
'T':'A', 'N':'N', 'a':'t', 'c':'g', 'g':'c', 't':'a',
'n':'n'}
>>> letters=list(dna)
>>> letters
['A', 'G', 'T', 'G', 'T', 'G', 'G', 'G', 'G', 'C', 'G']
>>> letters = [basecomplement[base] for base in
letters]
>>> letters
['T', 'C', 'A', 'C', 'A', 'C', 'C', 'C', 'C', 'G', 'C']
```

built-in `list` function returns a list whose items are the same and in the same order as the input sequence argument's items.

A Function To Complement A DNA Sequence

```
def complement(dna):
    """Return the complementary sequence
string."""
    basecomplement = {'A':'T', 'C':'G',
'G':'C', 'T':'A', 'N':'N', 'a':'t', 'c':'g',
'g':'c', 't':'a', 'n':'n'}
    letters = list(dna)
    letters = [basecomplement[base] for
base in letters]
    return ''.join(letters)
```

Split And Join

Split and *join* are methods of the string object.

Split. The method `split()` returns a list of all the words in the string:

```
>>> sentence="enzymes and other proteins come in many shapes"
>>> sentence.split()
['enzymes', 'and', 'other', 'proteins', 'come', 'in', 'many', 'shapes']
```

By default it splits on all whitespaces, but we can specify a separator:

```
>>> sentence.split('and')
['enzymes ', ' other proteins come in many shapes']
```

Join. The method `join()` returns a string in which the string elements were joined from a list. The separator string that joins the elements is the one upon which the function is called:

```
>>> '-'.join(['enzymes', 'and', 'other', 'proteins', 'come', 'in', 'many', 'shapes'])
'enzymes-and-other-proteins-come-in-many-shapes'
```

A Function To Complement A DNA Sequence

```
def complement(dna):
    """Return the complementary sequence
string."""
    basecomplement = {'A':'T', 'C':'G',
'G':'C', 'T':'A', 'N':'N', 'a':'t', 'c':'g',
'g':'c', 't':'a', 'n':'n'}
    letters = list(dna)
    letters = [basecomplement[base] for
base in letters]
    return ''.join(letters)
```

A Different Way To Complement A DNA Sequence

```
def complement(dna):

    """Return the complementary sequence string."""

    transtable=dna.maketrans('acgtnACGTN', 'tgcanTGCAN')

    return dna.translate(transtable)
```

returns a translation table where each character in first string will be mapped to the character at the same position in second string

returns a copy of the string where all characters have been mapped through the translation map created with the `maketrans()` method

Variable Number Of Function Arguments

Typical function declaration:

```
def myfunction(first, second, third):
    # do something with the 3 variables
    ...
```

It is possible to declare functions with a variable number of arguments, using the following syntax:

```
def newfunction(first, second, third, *therest):
            print("First: %s" % first)
            print("Second: %s" % second)
            print("Third: %s" % third)
            print("And all the rest... ",therest)
            return
```

```
>>> newfunction(1,2,3,4,5)
First: 1
Second: 2
Third: 3
And all the rest...  (4, 5)
```