

Python for scientific research

Functions and modules

Bram Kuijper

University of Exeter, Penryn Campus, UK

March 6, 2019



Researcher
Development



What we've done so far

- 1 Declare variables using built-in data types and execute operations on them
- 2 Use flow control commands to dictate the order in which commands are run and when
- 3 **Next:** Encapsulate programs into reusable functions, modules and packages

Motivation

- Imagine we wrote a series of commands to perform a particular task, for example, searching for a motif within a DNA sequence string

```
1 motif = "ggatcc" # sequence to search for
2 DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc" #
   my DNA
3 index = 0 # index of where to start looking for motif
4 indices = [] # result; list of indices where motif is
5 while index != -1: # -1 implies no match
6     index = DNA.find(motif, index)
7     if index != -1:
8         indices.append(index)
9         index += 1
```

Motivation

- Imagine we wrote a series of commands to perform a particular task, for example, searching for a motif within a DNA sequence string

```
1 motif = "ggatcc" # sequence to search for
2 DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc" #
   my DNA
3 index = 0 # index of where to start looking for motif
4 indices = [] # result; list of indices where motif is
5 while index != -1: # -1 implies no match
6     index = DNA.find(motif, index)
7     if index != -1:
8         indices.append(index)
9         index += 1
```

- We are now presented with a new DNA sequence and/or a different motif, what do we do?
 - 1 Copy and paste the above program and change `motif` and `DNA`

Motivation

- Imagine we wrote a series of commands to perform a particular task, for example, searching for a motif within a DNA sequence string

```
1 motif = "ggatcc" # sequence to search for
2 DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc" #
   my DNA
3 index = 0 # index of where to start looking for motif
4 indices = [] # result; list of indices where motif is
5 while index != -1: # -1 implies no match
6     index = DNA.find(motif, index)
7     if index != -1:
8         indices.append(index)
9         index += 1
```

- We are now presented with a new DNA sequence and/or a different motif, what do we do?
 - 1 Copy and paste the above program and change `motif` and `DNA`
 - 2 Encapsulate the commands into a reusable function

Anatomy of functions

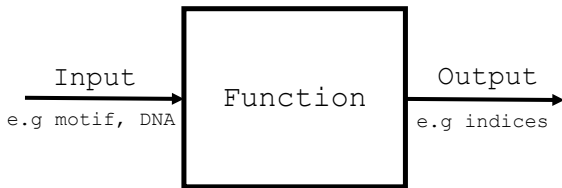
- Functions are ubiquitous in programming, enabling us to invoke the same function over and over again; **reusability**

Anatomy of functions

- Functions are ubiquitous in programming, enabling us to invoke the same function over and over again; **reusability**
- Using functions allow us to “hide” complexity (abstraction), making it easier to build complex programs, as we only need to worry about how to **use** the function rather than how it **works** on the inside

Anatomy of functions

- Functions are ubiquitous in programming, enabling us to invoke the same function over and over again; **reusability**
- Using functions allow us to “hide” complexity (abstraction), making it easier to build complex programs, as we only need to worry about how to **use** the function rather than how it **works** on the inside
- In a nutshell, functions take a number of **input** arguments (e.g DNA, motif) and return an **output** (e.g indices)



Simple functions

- Sum two numbers

```
1 def mysum(x, y):  
2     return x + y  
3  
4 # Call function  
5 out = mysum(10, 2) # out = 12
```

Simple functions

- Sum two numbers

```
1 def mysum(x, y):  
2     return x + y  
3  
4 # Call function  
5 out = mysum(10, 2) # out = 12
```

- Sum and divide two numbers

```
1 def sum_and_divide(x, y):  
2     return x+y, x/y  
3  
4 # Call function  
5 out1, out2 = sum_and_divide(10, 2) # out1 = 12, out2 =  
    5
```

Simple functions

- Sum, and divide two numbers after checking for division by zero

```
1 def sum_and_divide(x, y):
2     # Compute sum
3     mySum = x + y
4
5     # Compute division only if y is not zero
6     if y != 0:
7         myDiv = x/y
8     else:
9         myDiv = None
10
11     # Return sum and division results
12     return mySum, myDiv
13
14 # Call function
15 out1, out2 = sum_and_divide(10, 0) # out1 = 10, out2 =
    None
```

- Python functions lack { } used in many other languages (e.g R, C); **indentation** is everything!

- Python functions lack `{ }` used in many other languages (e.g R, C); **indentation** is everything!
- It is good practice to `return` something after a function call; if you don't Python will return an object of type `None`

- Python functions lack { } used in many other languages (e.g R, C); **indentation** is everything!
- It is good practice to `return` something after a function call; if you don't Python will return an object of type `None`
- Terminology
 - 1 **Parameters:** the variable names defined in the function definition
 - 2 **Arguments:** the values supplied to a function when it is called

Finding a motif within a DNA sequence

```
1 motif = "ggatcc" # sequence to search for
2 DNA = "acgtgtaaccaaggatccacccggttttaaacctgtgtgggatcc" # my
  DNA
3 index = 0 # index of where to start looking for motif
4 indices = [] # result; list of indices where motif is
5 while index != -1: # -1 implies no match
6     index = DNA.find(motif, index)
7     if index != -1:
8         indices.append(index)
9         index += 1
```

Wrap code into a function

```
1 def find_motif(DNA, motif):
2     index = 0 # index of where to start looking for motif
3     indices = [] # result; list of indices where motif is
4     while index != -1: # -1 implies no match
5         index = DNA.find(motif, index)
6         if index != -1:
7             indices.append(index)
8             index += 1
9     return indices # return an output; indices
```


Using default argument values

```
1 def find_motif(DNA, motif="gaatca"):
2     index = 0 # index of where to start looking for motif
3     indices = [] # result; list of indices where motif is
4     while index != -1: # -1 implies no match
5         index = DNA.find(motif, index)
6         if index != -1:
7             indices.append(index)
8             index += 1
9     return indices # return an output; indices
```

Always include a documentation string

```
1 def find_motif(DNA, motif="gaatca"):
2     """
3     Finds a motif within a DNA sequence and returns a list
4     of start indices
5
6     Parameters
7     -----
8     motif : a string to be matched
9     DNA : a string containing the DNA sequence to be
10           searched
11
12     Returns
13     -----
14     indices : list of start indices where motif is located
15     """
16     index = 0 # index of where to start looking for motif
17     indices = [] # result; list of indices where motif is
18     while index != -1: # -1 implies no match
19         index = DNA.find(motif, index)
20         if index != -1:
21             indices.append(index)
22             index += 1
23     return indices # return an output; indices
```

Calling functions

```
1 # Example
2 motif1 = "ggatcc" # sequence to search for
3 motif2 = "aacctg" # another sequence to search for
4 DNA = "acgtgtaaccaaggatccaccgttttaaacctgtgtgggatcc"
```

1 By argument order/position

```
1 indices1 = find_motif(DNA, motif1)
```

Calling functions

```
1 # Example
2 motif1 = "ggatcc" # sequence to search for
3 motif2 = "aacctg" # another sequence to search for
4 DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc"
```

1 By argument order/position

```
1 indices1 = find_motif(DNA, motif1)
```

2 By argument keyword (preferred)

```
1 indices2 = find_motif(motif=motif2, DNA=DNA)
```

Calling functions

```
1 # Example
2 motif1 = "ggatcc" # sequence to search for
3 motif2 = "aacctg" # another sequence to search for
4 DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc"
```

1 By argument order/position

```
1 indices1 = find_motif(DNA, motif1)
```

2 By argument keyword (preferred)

```
1 indices2 = find_motif(motif=motif2, DNA=DNA)
```

3 Using default arguments

```
1 indicesDefault = find_motif(DNA)
```

- We will typically write functions to perform a variety of related tasks

```
1 def complement(DNA):
2     """
3     Return the complement of a DNA sequence
4     """
5     <Your funky code>
6     return output
7
8 def reverse_complement(DNA):
9     """
10    Return the reverse complement of a DNA sequence
11    """
12    <Your funky code>
13    return output
14
15 def find_motif(motif, DNA):
16     """
17     Finds a motif within a DNA sequence
18     """
19     <Your funky code>
20     return output
21 ...
```

- **Modules** let us reuse functions in any program without the need to redefine them (read: copy and paste)

- **Modules** let us reuse functions in any program without the need to redefine them (read: copy and paste)
- Grouping functions by topic makes our code easier to use, understand and debug

- **Modules** let us reuse functions in any program without the need to redefine them (read: copy and paste)
- Grouping functions by topic makes our code easier to use, understand and debug
- **Modules** are simply Python files (.py) that contain definitions of functions and variables related to some specific theme

- **Modules** let us reuse functions in any program without the need to redefine them (read: copy and paste)
- Grouping functions by topic makes our code easier to use, understand and debug
- **Modules** are simply Python files (.py) that contain definitions of functions and variables related to some specific theme
- For example let us save the previously defined DNA sequence functions to a file called `dna_utils.py`; our new **module**

Importing modules

- We can access functions from modules by using the `import` command and `'.'` notation

```
1 # Preamble
2 import dna_utils
3
4 # Declare some variables
5 motif = "aacctg" # sequence to search for
6 DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" #
    my DNA
7
8 # Return complement of DNA sequence
9 compDNA = dna_utils.complement(DNA)
10
11 # Return reverse complement of DNA sequence
12 revCompDNA = dna_utils.reverse_complement(DNA)
13
14 # Find motif within DNA sequence
15 indices = dna_utils.find_motif(DNA, motif)
```

- What if I wrote the following modules?

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences
 - 2 `rna_utils.py`: functions for mRNA sequences

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences
 - 2 `rna_utils.py`: functions for mRNA sequences
 - 3 `protein_utils.py`: functions for protein coding sequences

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences
 - 2 `rna_utils.py`: functions for mRNA sequences
 - 3 `protein_utils.py`: functions for protein coding sequences
 - 4 `fasta_utils.py`: functions for FASTA files

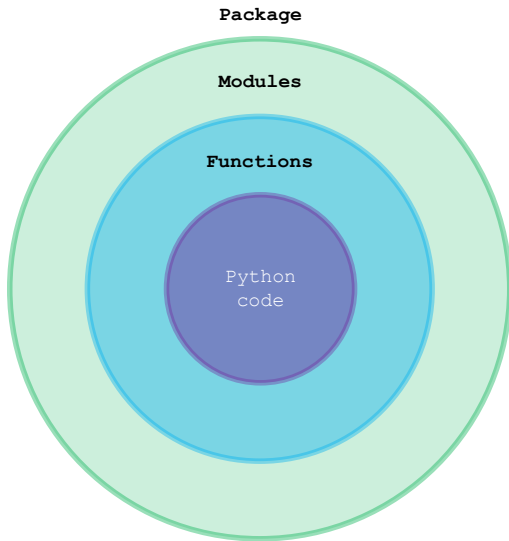
- What if I wrote the following modules?

- 1 `dna_utils.py`: functions for DNA sequences
- 2 `rna_utils.py`: functions for mRNA sequences
- 3 `protein_utils.py`: functions for protein coding sequences
- 4 `fasta_utils.py`: functions for FASTA files
- 5 `fastq_utils.py`: functions for FASTQ files
- 6 ...

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences
 - 2 `rna_utils.py`: functions for mRNA sequences
 - 3 `protein_utils.py`: functions for protein coding sequences
 - 4 `fasta_utils.py`: functions for FASTA files
 - 5 `fastq_utils.py`: functions for FASTQ files
 - 6 ...
- A **package** is a group of related modules that help us organise our code even further

- What if I wrote the following modules?
 - 1 `dna_utils.py`: functions for DNA sequences
 - 2 `rna_utils.py`: functions for mRNA sequences
 - 3 `protein_utils.py`: functions for protein coding sequences
 - 4 `fasta_utils.py`: functions for FASTA files
 - 5 `fastq_utils.py`: functions for FASTQ files
 - 6 ...
- A **package** is a group of related modules that help us organise our code even further
- A **package** is a normal folder containing the Python file `__init__.py` which tells Python that the folder contains modules

Hierarchical organisation: divide and conquer



Package example

- This is what our genomics package could look like

```
genomics/  
├── __init__.py  
├── dna_utils.py  
├── rna_utils.py  
├── protein_utils.py  
├── fasta_utils.py  
├── fastq_utils.py  
└── ...
```

```
1 import genomics.dna_utils  
2  
3 import genomics.rna_utils  
4  
5 import genomics.fasta_utils  
6  
7 ...
```

Package example

- Or we can organise it even further

```
genomics/  
├── __init__.py  
├── dna_utils.py  
├── rna_utils.py  
├── protein_utils.py  
├── fasta/  
│   ├── __init__.py  
│   ├── quality_control.py  
│   ├── read_write.py  
│   └── ...  
└── fastq/  
    ├── __init__.py  
    ├── quality_control.py  
    ├── read_write.py  
    └── ...
```

```
1 import genomics.dna_utils  
2  
3 import genomics.rna_utils  
4  
5 import genomics.fasta.  
    quality_control  
6  
7 import genomics.fasta.read_write  
8  
9 ...
```

Importing from a package

- We can access functions from modules in a package by using the `from ... import ...` command and `'.'` notation

```
1 # Preamble
2 from genomics import dna_utils
3
4 # Return complement of DNA sequence
5 compDNA = dna_utils.complement(DNA)
```

Importing from a package

- We can access functions from modules in a package by using the `from ... import ...` command and `'.'` notation

```
1 # Preamble
2 from genomics import dna_utils
3
4 # Return complement of DNA sequence
5 compDNA = dna_utils.complement(DNA)
```

- Going one level down the hierarchy

```
1 # Preamble
2 from genomics.fastq import quality_control
3
4 # Check if "sample1.fastq" is a valid FASTQ file
5 flag = quality_control.validate("sample1.fastq")
```


Importing from a package

- We can also import only the functions we need

```
1 # Preamble
2 from genomics.dna_utils import complement,
   reverse_complement
3
4 # Example
5 compDNA = complement(DNA) # complement
6 revCompDNA = reverse_complement(DNA) # reverse
   complement
```

Importing from a package

- We can also import only the functions we need

```
1 # Preamble
2 from genomics.dna_utils import complement,
   reverse_complement
3
4 # Example
5 compDNA = complement(DNA) # complement
6 revCompDNA = reverse_complement(DNA) # reverse
   complement
```

- Or rename the module/package upon importing

```
1 # Preamble
2 from genomics import dna_utils as util
3
4 # Example
5 compDNA = util.complement(DNA) # complement
6 revCompDNA = util.reverse_complement(DNA) # reverse
   complement
```

Warning

- We can import **all** functions and variables from a module as follows

```
1 # Preamble
2 from genomics.dna_utils import *
3
4 # Example
5 compDNA = complement(DNA) # complement
```

Warning

- We can import **all** functions and variables from a module as follows

```
1 # Preamble
2 from genomics.dna_utils import *
3
4 # Example
5 compDNA = complement(DNA) # complement
```

- **AVOID** using `import *`

Warning

- We can import **all** functions and variables from a module as follows

```
1 # Preamble
2 from genomics.dna_utils import *
3
4 # Example
5 compDNA = complement(DNA) # complement
```

- **AVOID** using `import *`
- “Explicit is better than implicit” - *The Zen of Python*

Warning

- We can import **all** functions and variables from a module as follows

```
1 # Preamble
2 from genomics.dna_utils import *
3
4 # Example
5 compDNA = complement(DNA) # complement
```

- **AVOID** using `import *`
- “**Explicit is better than implicit**” - *The Zen of Python*
- If you `import *` from several packages/modules you will get conflicts if functions have the same name

Warning

- We can import **all** functions and variables from a module as follows

```
1 # Preamble
2 from genomics.dna_utils import *
3
4 # Example
5 compDNA = complement(DNA) # complement
```

- **AVOID** using `import *`
- “**Explicit is better than implicit**” - *The Zen of Python*
- If you `import *` from several packages/modules you will get conflicts if functions have the same name
- “**Namespaces are one honking great idea**” - *The Zen of Python*

MATLAB®

- Python packages are equivalent to MATLAB toolboxes
- Toolboxes are loaded to the global namespace/workspace
- There's an equivalent `import` function introduced recently

MATLAB®

- Python packages are equivalent to MATLAB toolboxes
- Toolboxes are loaded to the global namespace/workspace
- There's an equivalent `import` function introduced recently

R

- Python packages are equivalent to R libraries/packages
e.g `library(tidyr)`
- Packages are loaded to the global namespace/workspace
- Using the double colon operator (`::`) conflicts can be avoided
e.g `tidyr::gather(...)`