Python for scientific research

Functions and modules

Bram Kuijper

University of Exeter, Penryn Campus, UK

February 11, 2020

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

```
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" #
    my DNA
index = 0 # index of where to start looking for motif
indices = [] # result; list of indices where motif is
while index != -1: # -1 implies no match
    index = DNA.find(motif, index)
    if index != -1:
        indices.append(index)
        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

```python
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" #
    my DNA
index = 0 # index of where to start looking for motif
indices = [] # result; list of indices where motif is
while index != -1: # -1 implies no match
    index = DNA.find(motif, index)
    if index != -1:
        indices.append(index)
        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

```python
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" #
    my DNA
index = 0 # index of where to start looking for motif
indices = [] # result; list of indices where motif is
while index != -1: # -1 implies no match
    index = DNA.find(motif, index)
    if index != -1:
        indices.append(index)
        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

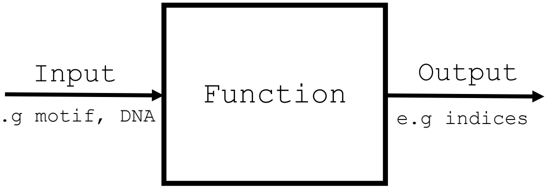
- Functions are ubiquitous in programming, enabling us to invoke the same function over and over again; **reusability**

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

```
# define the function
def mysum(x, y):
    return x + y

# call the function
out = mysum(10, 2) # out = 12
```

Simple functions

- Sum two numbers

```
# define the function
def mysum(x, y):
    return x + y

# call the function
out = mysum(10, 2) # out = 12
```

- Functions are first *defined* using `def` and then *called*

Simple functions

- Sum two numbers

```python
# define the function
def mysum(x, y):
    return x + y

# call the function
out = mysum(10, 2) # out = 12
```

- Functions are first *defined* using `def` and then *called*
- Functions can have one or more *parameters*; in this case `x` and `y`

Simple functions

- Sum two numbers

```python
# define the function
def mysum(x, y):
    return x + y

# call the function
out = mysum(10, 2) # out = 12
```

- Functions are first *defined* using `def` and then *called*
- Functions can have one or more *parameters*; in this case `x` and `y`
- Functions return a value (here `x + y`) to the “nonlocal scope” (aka the surrounding code calling the function)

Returning of multiple values

- Sum and divide two numbers: return multiple values

```
def sum_and_divide(x, y):
    return x+y, x/y

# Call function
out1, out2 = sum_and_divide(10, 2) # out1 = 12, out2 =
    5
```

Simple functions

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

```python
def sum_and_divide(x, y):
    # Compute sum
    mySum = x + y

    # Compute division only if y is not zero
    if y != 0:
        myDiv = x/y
    else:
        myDiv = None

    # Return sum and division results
    return mySum, myDiv

# Call function
out1, out2 = sum_and_divide(10, 0) # out1 = 10, out2 =
    None
```

Note

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

Note

- 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`

Note

- 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

```python
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" # my
    DNA
index = 0 # index of where to start looking for motif
indices = [] # result; list of indices where motif is
while index != -1: # -1 implies no match
    index = DNA.find(motif, index)
    if index != -1:
        indices.append(index)
        index += 1
```

Wrap code into a function

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

Using default argument values

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

Always include a documentation string

```python
def find_motif(DNA, motif="gaatca"):
    """
    Finds a motif within a DNA sequence and returns a list
    of start indices

    Parameters
    ----------
    motif : a string to be matched
    DNA : a string containing the DNA sequence to be
        searched

    Returns
    -------
    indices : list of start indices where motif is located
    """
    index = 0 # index of where to start looking for motif
    indices = [] # result; list of indices where motif is
    while index != -1: # -1 implies no match
        index = DNA.find(motif, index)
        if index != -1:
            indices.append(index)
            index += 1
    return indices # return an output; indices
```

Calling functions

```
# Example
motif1 = "ggatcc" # sequence to search for
motif2 = "aacctg" # another sequence to search for
focalDNA = "
    acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc"
```

1. By argument order/position

```
indices1 = find_motif(focalDNA, motif1)
```

Calling functions

```
# Example
motif1 = "ggatcc" # sequence to search for
motif2 = "aacctg" # another sequence to search for
focalDNA = "
    acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc"
```

1. By argument order/position

```
indices1 = find_motif(focalDNA, motif1)
```

2. By argument keyword (preferred)

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

Calling functions

```
# Example
motif1 = "ggatcc" # sequence to search for
motif2 = "aacctg" # another sequence to search for
focalDNA = "
    acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc"
```

1. By argument order/position

```
indices1 = find_motif(focalDNA, motif1)
```

2. By argument keyword (preferred)

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

3. Using default arguments

```
indicesDefault = find_motif(focalDNA)
# or better:
indicesDefault = find_motif(DNA=focalDNA)
```

Parameters and local variable scope

- Say, we define the following function

```python
def some_function(x):
    x += 1 # note: function returns nothing
```

Parameters and local variable scope

- Say, we define the following function

```python
def some_function(x):
    x += 1 # note: function returns nothing

a = 5
some_function(x=a) # call the function
print(a) # still 5
```

Parameters and local variable scope

- Say, we define the following function

```python
def some_function(x):
    x += 1 # note: function returns nothing

a = 5
some_function(x=a) # call the function
print(a) # still 5
```

- The parameter x only exists within the function body (i.e., *locally*)

Parameters and local variable scope

- Say, we define the following function

```python
def some_function(x):
    x += 1 # note: function returns nothing

a = 5
some_function(x=a) # call the function
print(a) # still 5
```

- The parameter `x` only exists within the function body (i.e., *locally*)
- When `some_function()` is called, `x` is assigned the reference to the value of `a`
- When `x` gets a new value, it gets a new reference to that new value
- When the function `some_function()` ends, all local variables are deleted

Parameters and local variable scope

- Say, we define the following function

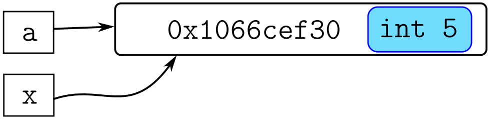
```python
def some_function(x):
    x += 1 # note: function returns nothing

a = 5
some_function(x=a) # call the function
print(a) # still 5
```

- The parameter `x` only exists within the function body (i.e., *locally*)
- When `some_function()` is called, `x` is assigned the reference to the value of `a`
- When `x` gets a new value, it gets a new reference to that new value
- When the function `some_function()` ends, all local variables are deleted

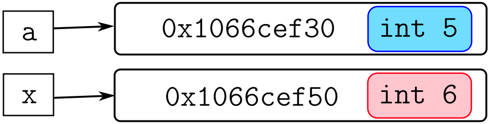
Parameters and local variable scope

- In graphics: what happens with variables during function calls

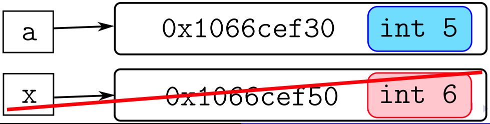
Function call: `some_function(x=a)`



Increment local variable: `x+=1`



Function ends:



Parameters and local variable scope

- Bottom line: for single-dimensional variables, references to values are destroyed when functions end

Parameters and local variable scope

- Bottom line: for single-dimensional variables, references to values are destroyed when functions end
- Updating values used outside the function (i.e., “nonlocal” values) should be done through the `return` statement:

Parameters and local variable scope

- Bottom line: for single-dimensional variables, references to values are destroyed when functions end
- Updating values used outside the function (i.e., “nonlocal” values) should be done through the `return` statement:

```python
def some_function(x):
    x += 1
    return x

a = 5
a = some_function(x=a) # call the function
print(a) # 6
```

Modules

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

```python
def complement(DNA):
    """
    Return the complement of a DNA sequence
    """
    <Your funky code>
    return output

def reverse_complement(DNA):
    """
    Return the reverse complement of a DNA sequence
    """
    <Your funky code>
    return output

def find_motif(motif, DNA):
    """
    Finds a motif within a DNA sequence
    """
    <Your funky code>
    return output
...
```

Modules

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

Modules

- **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

- **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

- **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

```python
# Preamble
import dna_utils

# Declare some variables
motif = "aacctg" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgggatcc" #
    my DNA

# Return complement of DNA sequence
compDNA = dna_utils.complement(DNA)

# Return reverse complement of DNA sequence
revCompDNA = dna_utils.reverse_complement(DNA)

# Find motif within DNA sequence
indices = dna_utils.find_motif(DNA, motif)
```

Packages

- What if I wrote the following modules?

Packages

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

Packages

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

Packages

- 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

Packages

- 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

Packages

- 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. ...

Packages

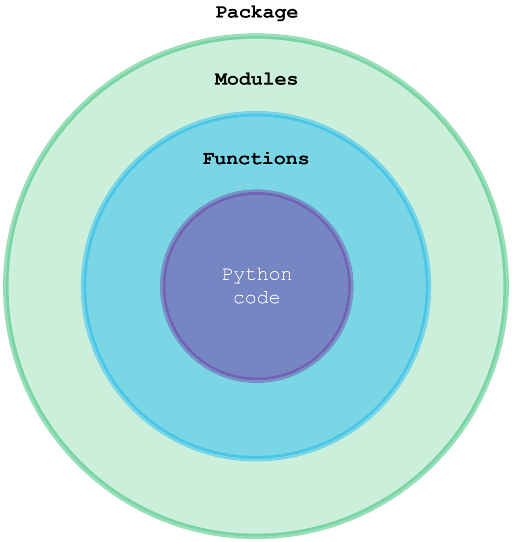
- What if I wrote the following modules?
  1. `dna_utils.py`: functions for DNA sequences
  2. `rna_utils.py`: functions for mRNA sequences
  3. `protein_utils.py`: functions for protein coding sequences
  4. `fasta_utils.py`: functions for FASTA files
  5. `fastq_utils.py`: functions for FASTQ files
  6. ...

- A **package** is a group of related modules that help us organise our code even further

Packages

- 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
└── ...
```

```python
import genomics.dna_utils

import genomics.rna_utils

import genomics.fasta_utils

...
```

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
    └── ...
```

```python
import genomics.dna_utils

import genomics.rna_utils

import genomics.fasta.
    quality_control

import genomics.fasta.read_write

...
```

Importing from a package

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

```python
# Preamble
from genomics import dna_utils

# Return complement of DNA sequence
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

```
# Preamble
from genomics import dna_utils

# Return complement of DNA sequence
compDNA = dna_utils.complement(DNA)
```

- Going one level down the hierarchy

```
# Preamble
from genomics.fastq import quality_control

# Check if "sample1.fastq" is a valid FASTQ file
flag = quality_control.validate("sample1.fastq")
```

Importing from a package

- We can also import only the functions we need

```python
# Preamble
from genomics.dna_utils import complement,
    reverse_complement

# Example
compDNA = complement(DNA) # complement
revCompDNA = reverse_complement(DNA) # reverse
    complement
```

Importing from a package

- We can also import only the functions we need

```
# Preamble
from genomics.dna_utils import complement,
    reverse_complement

# Example
compDNA = complement(DNA) # complement
revCompDNA = reverse_complement(DNA) # reverse
    complement
```

- Or rename the module/package upon importing

```
# Preamble
from genomics import dna_utils as util

# Example
compDNA = util.complement(DNA) # complement
revCompDNA = util.reverse_complement(DNA) # reverse
    complement
```

Warning

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

```
# Preamble
from genomics.dna_utils import *

# Example
compDNA = complement(DNA) # complement
```

Warning

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

```python
# Preamble
from genomics.dna_utils import *

# Example
compDNA = complement(DNA) # complement
```

- **AVOID** using `import *`

Warning

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

```
# Preamble
from genomics.dna_utils import *

# Example
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

```python
# Preamble
from genomics.dna_utils import *

# Example
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

```python
# Preamble
from genomics.dna_utils import *

# Example
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*

Final note

MATLAB®

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

Final note

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(...)`