

# Python for scientific research

## Functions, modules and packages

John Joseph Valletta

University of Exeter, Penryn Campus, UK

June 2017



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

```
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccgttttaaacctgtgtgaggatcc" # 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?

# 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
```

- 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

# 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 = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc" # 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

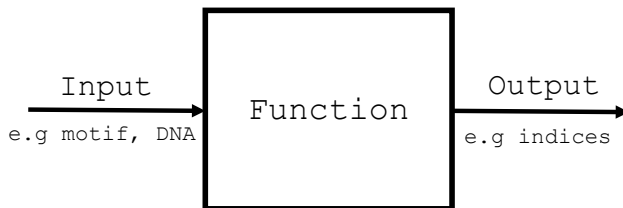
- Functions are ubiquitous in programming, enabling us to invoke the same function over and over again; **reusability**
- Using functions allow us to “hide” complexity (abstraction), making it easier to build complex programs, as we only need to worry about how to **use** the function rather than how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

# 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)

# 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

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

- Sum and divide two numbers

```
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 two numbers

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

- Sum and divide two numbers

```
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

```
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
```

- 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

- 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

- 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

```
motif = "ggatcc" # sequence to search for
DNA = "acgtgtaaccaaggatccacccggttttaaacctgtgtgggatcc" # 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

```
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

```
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 with your function

```
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
DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc"
```

## ① By argument order/position

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

## ② By argument keyword

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

## ③ Using default arguments

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

# Calling functions

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

## ① By argument order/position

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

## ② By argument keyword

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

## ③ Using default arguments

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

# Calling functions

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

## 1 By argument order/position

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

## 2 By argument keyword

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

## 3 Using default arguments

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

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

```
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** 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**

- **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**



- **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**

- **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

```
# Preamble
import dna_utils

# Declare some variables
motif = "aacctg" # sequence to search for
DNA = "acgtgtaaccaaggatccaccggttttaaacctgtgtgggatcc" # 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)
```

- 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

- 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

- 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

- 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

- 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

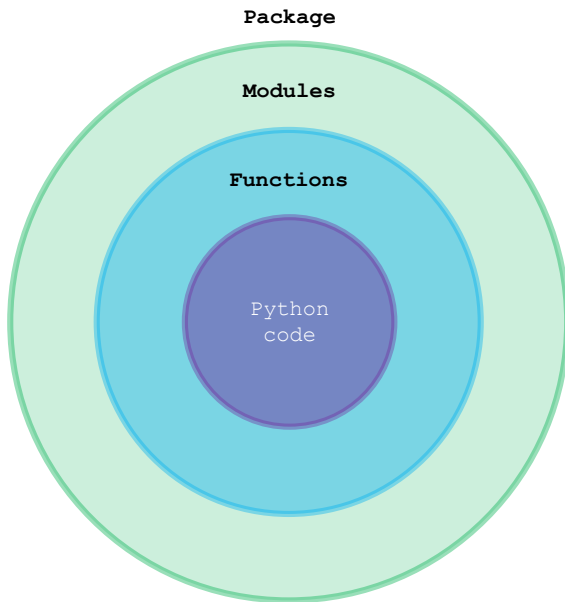


- What if I wrote the following modules?
  - 1 `dna_utils.py`: functions for DNA sequences
  - 2 `rna_utils.py`: functions for mRNA sequences
  - 3 `protein_utils.py`: functions for protein coding sequences
  - 4 `fasta_utils.py`: functions for FASTA files
  - 5 `fastq_utils.py`: functions for FASTQ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

- 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

- 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  
└── ...
```

```
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  
    └── ...
```

```
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

```
# 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 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

```
# 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
```

# 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
```

- **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*

# 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*
  - 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*

# 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*
- 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*

# 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*
- 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*

# 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*
- 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(...)`