Foundations of Data Science for Biologists

Flow control and functions Python

BIO 724D

2024-MAR-18

Instructors: Greg Wray and Paul Magwene

Control flow

Control flow in Python

Python provides `if`, `for`, and `while` control flow structures

- The basic syntax is similar to R, although a few additional options are available
- Test conditions are expressed in a similar way to R
- Nesting and mixed nesting of structures is allowed

All three control flow structures can optionally include a `finally` statement (unlike R)

- Executes once if the preceding structure/loop finishes without any `break` or error
- Useful for reporting successful completion or for passing warnings

In addition, Python provides idioms for iterables that avoid explicit loops

- Iterable comprehensions, the most common being list comprehension
- Conditional assignment / ternary conditionals

Special control flow statements

Three special statements can be included in control flow structures

- None are necessary (work-arounds exist) but they clarify logic and are more readable

`pass`: a valid statement that does nothing

- Useful as a placeholder for later work (blocks must contain at least one statement)

`continue`: proceeds directly to the next iteration of a loop or next evaluation in an if structure

- Useful if a computationally intensive step doesn't need to execute each iteration
- Ignores all subsequent statements in the code block for that iteration *only*

`break`: completely exits the current loop and proceeds to the next statement

- Useful when searching an iterable for a single specific item
- Only works with loop structures; ignores *all* subsequent statements and conditions
- Only applies to the current loop structure (not to an enclosing loop, if present)
- Not to be confused with `return`, which performs the analogous role in a function

If structures

At most one conditional code block executes prior to the `finally` block (if present)

```
if <condition> :              # conditional execution
     <statement(s)>

if <condition> :              # alternative execution
     <statement(s)>
else :
     <statement(s)>

if <condition> :              # chained conditionals
     <statement(s)>
elif :                           # an arbitrary number of elifs is allowed
    <statement(s)>
else :
     <statement(s)>
```

If structures with finally

```
if <condition> :                # alternative execution
     <statement(s)>
else :
     <statement(s)>
finally :                       # always executes
     <statement(s)>

if <condition> :                # chained conditionals
     <statement(s)>
elif :                          # an arbitrary number of elifs is allowed
    <statement(s)>
else :
     <statement(s)>
finally :                       # always executes
     <statement(s)>
```

Multiple conditions

When testing multiple conditions, order matters for two reasons

Logic

- Conditions are evaluated in the order they are encountered
- The first conditional that evaluates to True ends the if / elif part of the structure
- Remember: the code block of *at most* one condition is evaluated

Efficiency

- Conditions are evaluated in the order they are encountered
- Test conditions in the order of expected frequency to minimize execution time

Conditional assignment

Python provides special if / else structure for conditional assignment

- Syntax is simpler and (often) easier to read, but reserve for simple conditions
- Known as conditional assignment or ternary operator

A normal if / else structure used for assignment looks like this:

```
if <condition> :
    my_var = <statement>
else :
    my_var = <alternative statement>
```

The syntax for a ternary operator:

```
my_var = <statement> if <condition> else <alternative statement>
```

An example to illustrate usage:

```
my_var = b if b > 10 else c   # assign b unless b > 10, in which case assign c
```

For loops

For loops use an iterable to determine the number of times the loop executes

The loop structure automatically updates the loop counter

Nesting for loops is a common way to systematically query a matrix or data frame

```
for <var> in <iterable> :                    # simple loop
    <statement(s)>

for <var> in <iterable> : <statement>    # one-line (only for 1 statement)

for <var> in <iterable> :                    # loop followed by completion block
    <statement(s)>
finally :                                    # always executes
    <statement(s)>
```

Using ranges with for loops

The `range()` function provides a simple way to create custom iterators

The syntax is:

```
for <var> in range(i, j, k) :   # can take 1, 2, or 3 arguments (see below)
    <statement(s)>
```

Examples to illustrate the usage of `range()`:

```
for i in range(6) :             # iterates 6 times, i takes values 0 to 6
for i in range(2, 6) :          # iterates 4 times, i takes values 2 to 5
for i in range(0, 6, 2) :       # iterates 4 times, i takes values 0, 2, 4, 6
```

Comprehensions

Python provides a functional programming idiom to specify for loops

A standard for loop that operates on a list looks like this:

```
fruits = [‘banana’, ‘pineapple’, ‘plum’]
new_list = []
for i in fruits :
    if ‘p’ in i : new_list.append()
```

A list comprehension that does the same think looks like this:

```
fruits = [‘banana’, ‘pineapple’, ‘plum’]
new_list = [i for i in fruits if ‘p’ in i]
```

While loops

While loops repeatedly execute a code block until a condition evaluates to `False`

- Will not execute even once if the condition evaluates to `False` on first iteration

Brings the risk of infinite looping if the condition never evaluates to `False`

- Ensuring termination may require declaring relevant condition variable(s) in advance
- Not generally recommended: only use if no other options work well

```
while <condition> :                    # simple loop
    <statement(s)>

while <condition> :                    # loop followed by completion block
    <statement(s)>
finally :
    <statement(s)>                     # always executes
```

Ending while loops

Best practice with while loops: include a mechanism to guarantee exiting the loop

Strategy: set up conditions that *inevitably* trigger a `break` statement

Many approaches can be used; 2 examples are shown below

```
i = 0                                   # using a loop counter
while <condition> :
     <statement(s)>
    i += 1
    if i > 1000 : break                     # exit if 1000 loops

while <condition> :                     # using user action
     <statement(s)>
    a = input('press any key to proceed')
    if a : break                            # user can interrupt at will
```

Functions

The basics

What does a function do?

- Takes input and produces an output
- Can be treated as a “black box” from a logic perspective
  - All the programmer cares about is input and output
  - Specifics of how the transformation works is not important
  - Contributes to abstraction

Types of functions available in Python

- Built-in: available without import or declaration
- Standard: must first load the module or function; huge variety of useful functions
- User-defined: must first declare
- Anonymous: single-use, single-line; also known as lambda functions

Arguments

Technically

Parameters are stated in the function declaration

Arguments are passed to the parameters during run-time

Python is flexible, such that functions may take:

0, 1, or multiple arguments (no upper bound)

A fixed or arbitrary number of arguments

Required arguments, optional arguments, or both

Any object as an argument, including other functions

Return values

By default, every function returns the value `None`

This behavior can be altered using a `return` statement

- 0, 1, or multiple values can be returned (no upper bound)
- Best practice is to place `return` statements at the end of the code block
- Multiple `return` statements can be included when using if / else structures

Functions can return values or named data objects

Defining a function

Creating a function

Use the `def` keyword

Use colon and indenting rather than curly braces to define the code block

```
# create a function to test whether 7 is a factor
def is_div_by_7(input_value) :
    if input_value % 7 == 0 : return True
    else: return False
# call the function
result_1 = is_div_by_7(3)
result_2 = is_div_by_7(42)
print(result_1, result_2)                    # returns False True
```

Defining a function, continued

Functions must contain at least statement in the code block

- Use pass as a place-holder to fill the code block (does nothing during execution)

Use return statements to control what the function returns

- If there is no return statement, the function returns the value `None`
- Using return by itself also returns the value `None` (useful in if/else structures)
- Using return followed by value(s) returns the value(s)

Use a **docstring** to provide a short description of the function and its use

- Place in triple quotes on the line immediately following the function declaration

```
"""Takes a DNA string as input and returns codons as a list"""
```

Defining arguments for functions

Optional arguments are indicated with = followed by the default value

```
def function_name(arg1 = 0, arg2 = True) :
    <statement(s)>
```

Optional arguments must follow all required arguments

The ability to pass an arbitrary number of arguments is indicated with `*args`

```
def function_name(*args) :
    <statement(s)>
```

Represented internally as a tuple; access as `args` within the function

The ability to pass an arbitrary number of keyword arguments is indicated with `**kwargs`

```
def function_name(**kwargs) :
    <statement(s)>
```

Represented internally as a dictionary; access as `kwargs` within the function

Passing arguments to functions

Separate multiple arguments with commas

Required arguments

- Pass these values first and in the defined order
- Anything else will throw an error

*args and **kwargs arguments

- Pass these values after any required arguments

Optional arguments

- Pass these last; order is not important
- Omit unless changing the default value (including with the default value is innocuous)
- Include the argument and value (e.g., `reverse = True`)

Scope

Global variables can be updated within a function

- However, this will not alter the value of the global variable *outside* the function

Any variable defined within a function is assumed to be local to that function

To define a global variable within a function, use the `global` keyword (not recommended)

```
def my_func() :
    global my_var      # first, declare the variable as global
    my_var = 'spam'    # then, assign a value
```

To return all variables and values within a function, call `locals()` or `globals()`

```
def my_func() :
    locals().          # returns a dictionary of local variables and values
    globals()          # returns a dictionary of global variables and values
```

Evaluation

Functions are not evaluated when defined

- Only evaluated when called by other code at runtime (known as lazy evaluation)

Default argument values are evaluated once, left to right, when the function is executed

- Can cause unexpected behavior if a default value is altered within the function

Keep both points in mind when debugging code!