

Python for scientific research

Flow control

Bram Kuijper

University of Exeter, Penryn Campus, UK

February 11, 2020



Researcher
Development



What we've done so far

- 1 Declare variables using built-in data types and execute operations on them
- 2 **Next:** Controlling the flow of a program

- Executing code one line at a time is useful but limiting

- Executing code one line at a time is useful but limiting
- **Flow control** commands lets us dictate the **order** in which commands are run:

- Executing code one line at a time is useful but limiting
- **Flow control** commands lets us dictate the **order** in which commands are run:
 - 1 **If-else**: to change what commands are executed under certain conditions

- Executing code one line at a time is useful but limiting
- **Flow control** commands lets us dictate the **order** in which commands are run:
 - 1 **If-else**: to change what commands are executed under certain conditions
 - 2 **For loops**: to repeat the same thing N times

- Executing code one line at a time is useful but limiting
- **Flow control** commands lets us dictate the **order** in which commands are run:
 - 1 **If-else**: to change what commands are executed under certain conditions
 - 2 **For loops**: to repeat the same thing N times
 - 3 **While loops**: to repeat the same thing until a specific condition is met

- Print whether the integer x is positive, negative or zero

```
1 if x > 0:
2     print("x is positive")
3 elif x < 0:
4     print("x is negative")
5 else:
6     print("x is zero")
```

- Note the lack of `{ }` used in many other languages (R, C/C++); in Python **indentation** is everything!
- Indent by using 4 spaces per indentation level, rather than tabs (see [PEP-008](#))
- Code indented using a mixture of tabs and spaces does not run

For loops

- Print the integers 1 to 5

```
1 for x in range(5):  
2     print(x+1)
```

For loops

- Print the integers 1 to 5

```
1 for x in range(5):  
2     print(x+1)
```

- Loop through a list of gene names and print them in upper case

```
1 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10"  
               ""]  
2 for gene in geneNames:  
3     print(gene.upper())
```

While loops

- Print the integers 10 to 1

```
1 x = 10
2 while x > 0:
3     print(x)
4     x = x - 1
```

- **Note:**

- 1 Use for loops over while loops where possible
- 2 Ensure that the while condition evaluates to False at some point to avoid an infinite loop

List Comprehensions

- List comprehensions are an optimized and readable method for creating a list

List Comprehensions

- List comprehensions are an optimized and readable method for creating a list
- Recall the previous example where we looped over gene names and printed them in upper case

```
1 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10"]  
2 for gene in geneNames:  
3     print(gene.upper())
```

List Comprehensions

- List comprehensions are an optimized and readable method for creating a list
- Recall the previous example where we looped over gene names and printed them in upper case

```
1 geneNames = ["Irf1", "Ccl3", "Il12rb1", "Ifng", "Cxcl10"]
2 for gene in geneNames:
3     print(gene.upper())
```

- What if I want to store the upper case gene names in another variable, called `x` for simplicity?

List Comprehensions

① Using for loops:

```
1 x = [] # create an empty list to append to
2 for gene in geneNames:
3     x.append(gene.upper())
```

List Comprehensions

1 Using for loops:

```
1 x = [] # create an empty list to append to
2 for gene in geneNames:
3     x.append(gene.upper())
```

2 Using list comprehension:

```
1 x = [gene.upper() for gene in geneNames]
```


List Comprehensions

1 Using for loops:

```
1 x = [] # create an empty list to append to
2 for gene in geneNames:
3     x.append(gene.upper())
```

2 Using list comprehension:

```
1 x = [gene.upper() for gene in geneNames]
```

3 What if I want to ignore gene Ifng?

```
1 x = [gene.upper() for gene in geneNames if gene != "
    Ifng"]
```

Enumerate

- When looping over lists, sometimes it's useful to keep track of the index of the iteration

Enumerate

- When looping over lists, sometimes it's useful to keep track of the index of the iteration
- **Enumerate** is a built-in function that lets us access the iterable element but also its index

Enumerate

- When looping over lists, sometimes it's useful to keep track of the index of the iteration
- **Enumerate** is a built-in function that lets us access the iterable element but also its index
- Print the index next to upper cased gene name
 - ① Using a standard for loop:

```
1 i = 0 # index counter
2 geneNames = ["a","b","c"]
3 for gene in geneNames:
4     print(f"Gene {i+1} is {gene.upper()}")
5     i = i + 1
```

Enumerate

- When looping over lists, sometimes it's useful to keep track of the index of the iteration
- **Enumerate** is a built-in function that lets us access the iterable element but also its index
- Print the index next to upper cased gene name
 - 1 Using a standard for loop:

```
1 i = 0 # index counter
2 geneNames = ["a","b","c"]
3 for gene in geneNames:
4     print(f"Gene {i+1} is {gene.upper()}")
5     i = i + 1
```

- 2 Using enumerate:

```
1 for i, gene in enumerate(geneNames):
2     print(f"Gene {i+1} is {gene.upper()}")
```

Iterate over a dictionary

- In case of a **dictionary** there are both keys and values to iterate over
- Use the **items()** method to iterate over list value pairs:

```
1 someDictionary = {"a" : "value1", "b" : "value2", "c" :  
    "value3"}  
2  
3 for key, value in someDictionary.items():  
4     print(f"{key} points at {value}")  
5  
6 # prints:  
7 # a points at value1  
8 # b points at value2  
9 # c points at value3
```

Quitting loops using break

- With a `break` statement you can end a `for` or `while` loop
- Statements within the loop that occur after a `break` occurs will not be executed and the loop terminates

```
1 for key, value in someDictionary.items():  
2     # end the loop immediately when  
3     # a certain condition is met  
4     if key == "b":  
5         break  
6  
7     # won't be executed when condition for break is met  
8     print(f"{key} points at {value}")
```

Quitting loops using break

- With a `break` statement you can end a `for` or `while` loop
- Statements within the loop that occur after a `break` occurs will not be executed and the loop terminates

```
1 for key, value in someDictionary.items():
2     # end the loop immediately when
3     # a certain condition is met
4     if key == "b":
5         break
6
7     # won't be executed when condition for break is met
8     print(f"{key} points at {value}")
9
10 # prints:
11 # a points at value1
```

- There is more to `break` clauses than fits this course: have a look at `for...else` clauses if you are interested.

Go to the next iteration with `continue`

- A `continue` statement makes the loop move on to the next iteration (by moving on to execute the statement at the top of the loop)
- Statements within the loop that occur after a `continue` will not be executed *during the current iteration of the loop*

```
1 for key, value in someDictionary.items():
2     # continue to the next iteration
3     # when a certain condition is met
4     if key == "b":
5         continue # go to for... in line 1
6
7     print(f"{key} points at {value}")
8
9
10 # output:
11 # a points at value1
12 # c points at value3
13 # (the item with the "language" label is not printed)
```

Nested loops

- Looping through multidimensional objects requires nested loops

```
1 # 2-dimensional list with 2 rows, 3 columns
2 mlist = [[1, 2, 3], [4, 5, 6]]
```

Nested loops

- Looping through multidimensional objects requires nested loops

```
1 # 2-dimensional list with 2 rows, 3 columns
2 mlist = [[1, 2, 3], [4, 5, 6]]
3
4 # access all individual elements through nested loop:
5 # loop over rows with
6 for row in mlist:
7     # loop over items in row
8     for col_item in row:
9         print(col_item)
10
11 # print:
12 # 1
13 # 2
14 # etc
```