Practical Programming
in Python

Inspired by 'Practical Programming' by Paul Gries, Jennifer Campbell, Jason Montojo

Lecture 9
Repeating Code: Loops

Loops, Iteration, Nested Loops, Controlling Loops

Kurt Rinnert, Kate Shaw

Physics Without Frontiers

The Abdus Salam International Centre for Theoretical Physics

Abstract

"In theory, practice is simple."
– Trygve Reenskaug

This lecture introduces a very important kind of control flow: repetition.
We don't want to write the same expression or instruction hundreds of times in our programs – that's clearly a job for a machine.
Now we'll learn how to write them once and use *loops* to repeat instructions and how to control when to stop the repetition.

Overview

- Processing items in a list
- Processing characters in strings
- Ranges of numbers
- The concept of iteration
- Nesting loops
- Controlling loops

Without repetition – loops – programs would not be very useful.

Processing Items in a List

- With what what we have learned so far, we would need to access list items one by one:

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> print('Metric:', velocities[0], 'm/sec;', 'Imperial:', velocities[0] * 3.28, 'ft/sec')
Metric: 0.0 m/sec; Imperial: 0.0 ft/sec
>>> print('Metric:', velocities[1], 'm/sec;', 'Imperial:', velocities[1] * 3.28, 'ft/sec')
Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec
>>> print('Metric:', velocities[2], 'm/sec; ', 'Imperial:', velocities[2] * 3.28, 'ft/sec')
Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec
>>> print('Metric:', velocities[3], 'm/sec; ', 'Imperial:', velocities[3] * 3.28, 'ft/sec')
Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec
```

We clearly don't want to do this for thousands of values.

Processing Items in a List

- Python provides the **`for`** loop that lets you process list elements:

```
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for velocity in velocities:
...     print('Metric:', velocity, 'm/sec;', 'Imperial:', velocity * 3.28, 'ft/sec')
...
Metric: 0.0 m/sec; Imperial: 0.0 ft/sec
Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec
Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec
Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec
```

Now we can process all items with one statement.

Processing Items in a List

- The general form of a **`for`** loop is:

```
for variable in list:
    block
```

- The *loop variable* is assigned to the first item in the list, and the *loop block* – the *body* of the **`for`** loop – is executed
- The loop variable is then assigned the second item in the list and the loop body is executed again

  …
- Finally, the loop variable is assigned the last item of the list and the loop body is executed one last time

Each execution of the loop body is an *iteration*.

Looping Over Items in List `velocities`

Iteration	Loop variable value	Output String
1st	`velocities[0]`	`'Metric: 0.0 m/sec; Imperial: 0.0 ft/sec'`
2nd	`velocities[1]`	`'Metric: 9.81 m/sec; Imperial: 32.1768 ft/sec'`
3rd	`velocities[2]`	`'Metric: 19.62 m/sec; Imperial: 64.3536 ft/sec'`
4th	`velocities[3]`	`'Metric: 29.43 m/sec; Imperial: 96.5304 ft/sec'`

Processing Items in a List

- It is possible to use a previously defined variable as the loop variable:

```
>>> speed = 2
>>> velocities = [0.0, 9.81, 19.62, 29.43]
>>> for speed in velocities:
...     print('Metric:', speed, 'm/sec')
...
Metric: 0.0 m/sec
Metric: 9.81 m/sec
Metric: 19.62 m/sec
Metric: 29.43 m/sec
```

- The loop variable keeps its value after the last iteration:

```
>>> print('Final:', speed)
Final: 29.43
```

Note that the last `print` statement is not part of the loop.

Processing Characters in Strings

- We can also loop over characters in a string
- The general form is:

```
for variable in str:
    block
```

- For example, we can loop over a string an print the uppercase characters:

```
>>> theory = 'Quantum Field Theory'
>>> for ch in theory:
...     if ch.isupper():
...         print(ch)
...
Q
F
T
```

How many iterations are in this loop?

Ranges of Numbers

- Python provides the built-in function `range` that generates a sequence of integers
- With a single argument, as in `range(stop)`, the sequence starts at `0` and ends with `stop - 1`

```
>>> range(4)
range(0, 4)
```

- We can loop over this sequence:

```
>>> for num in range(4):
...     print(num)
0
1
2
3
```

We can loop over all kinds of sequences – *iterables* – in Python.

Ranges of Numbers: Converting to Lists

- We can construct a `list` from a `range`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Here are some more examples:

```
>>> list(range(3))
[0, 1, 2]
>>> list(range(1))
[0]
>>> list(range(0))
[]
```

Note that `range` specifications are consistent with indexing and slicing.

Ranges of Numbers: Changing the Lower Bound

- We can also pass two arguments to the `range` function:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

By default, the *step size* is one.

Ranges of Numbers: Changing the Step Size

- The step size can be specified with a third argument
- Here is a list of the leap years in the first half of the 21st century:

```
>>> list(range(2000, 2050, 4))
[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032, 2036, 2040, 2044, 2048]
```

- Descending sequences can be produced with negative step sizes:

```
>>> list(range(2048, 1999, -4))
[2048, 2044, 2040, 2036, 2032, 2028, 2024, 2020, 2016, 2012, 2008, 2004, 2000]
```

What do you think this expression produces: `list(range(2, 8, -2))`?

Processing Lists Using Indices

- What if we want to change the items in a `list`?
- For example, we might want to double all values in a `list`
- The following does not work:

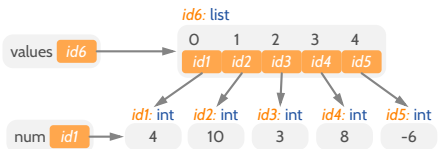
```
>>> values = [4, 10, 3, 8, -6]
>>> for num in values:
...     num *= 2
...
>>> values
[4, 10, 3, 8, -6]
```

Why does this not work as intended?

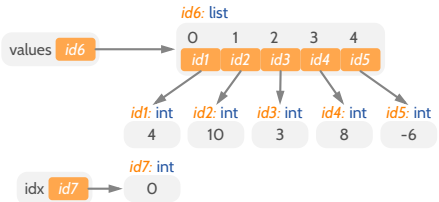
Processing Lists Using Indices

- Iteration is on the left, indexing on the right
- This is the memory model when the loop starts:

```
  >>> values = [4, 10, 3, 8, -6]
► >>> for num in values:
...         num *= 2
...
```




```
  >>> values = [4, 10, 3, 8, -6]
► >>> for idx in range(len(values)):
...         values[idx] *= 2
...
```

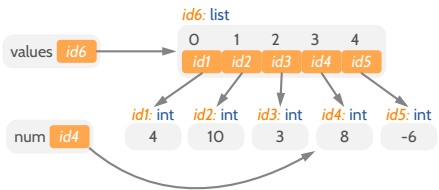


Note that `num` is referring to a value in the list while `idx` is not.

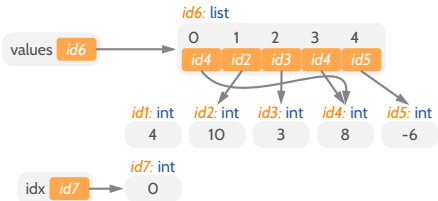
Processing Lists Using Indices

- Iteration is on the left, indexing on the right
- This is the memory model after the first iteration:

```
>>> values = [4, 10, 3, 8, -6]
>>> for num in values:
...     num *= 2
...
```




```
>>> values = [4, 10, 3, 8, -6]
>>> for idx in range(len(values)):
...     values[idx] *= 2
...
```

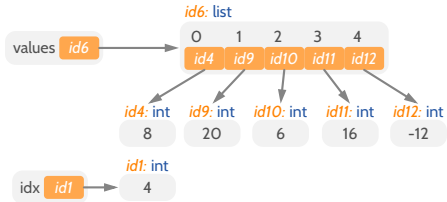


Doubling `num` in the loop body does not mutate the list.

Processing Lists Using Indices

- Let's see what happens when all iterations are done:

```
>>> values = [4, 10, 3, 8, -6]
>>> for idx in range(len(values)):
...     values[idx] *= 2
...
... print(values)
[8, 20, 6, 16, -12]
```



All elements have been doubled. The list has been mutated.

Processing Parallel Lists Using Indices

- Sometimes data from one list corresponds to data from another
- For example, consider these two lists:

```
>>> metals = ['Li', 'Na', 'K']
>>> weights = [6.941, 22.98976928, 39.0983]
```

- We can process them in parallel using indices:

```
>>> for i in range(len(metals)):
...     print(metals[i], weights[i])
...
Li 6.941
Na 22.98976928
K 39.0983
```

We will learn about more elegant ways of doing this later.

Nesting Loops in Loops

- The loop block can contain another loop
- In the code on the right, the inner loop is executed once for each item in `outer`
- The `print` function is called `len(outer)*len(inner)` times

```
>>> outer = ['Li', 'Na', 'K']
>>> inner = ['F', 'Cl', 'Br']
>>> for metal in outer:
...     for halogen in inner:
...         print(metal + halogen)
...
...
LiF
LiCl
LiBr
NaF
NaCl
NaBr
KF
KCl
KBr
```

Nesting often indicates something complicated is going on.

Nesting Loops in Loops

- Inner and outer loops can use the same list
- For example, in the code on the right
- Each outer iteration prints a row
- Let's have a look at the third iteration:

1. `i` is assigned 3, the third item in `numbers`
2. The row number, 3, is printed
3. This inner loop header is executed once per outer iteration
4. The inner loop body is executed five times. The first time it prints 3, then 6, and so on
5. A newline is printed after the row is completed

Avoid modifying lists in these scenarios.

```
def print_table(n):
    """
    Print multiplication table.

    Print the multiplication table
    for numbers 1 through n inclusive.

    Examples:

        >>> print_table(5)
            1       2       3       4       5
        1   1       2       3       4       5
        2   2       4       6       8       10
        3   3       6       9       12      15
        4   4       8       12      16      20
        5   5       10      15      20      25
    """
    # The numbers to include in the table.
    numbers = list(range(1, n + 1))

    # Print header
    for i in numbers:
        print('\t' + str(i), end='')
    print()

    # Print the numbered table rows
1   for i in numbers:
2       print(i, end='')
3       for j in numbers:
4           print('\t' + str(i * j), end='')
5       print()
```

Looping Over Nested Lists

- We can also loop over lists in lists (*nested* lists):

```
>>> elements = [[ 'Li', 'Na', 'K'], ['F', 'Cl', 'Br']]
>>> for inner_list in elements:
...     print(inner_list)
...
['Li', 'Na', 'K']
['F', 'Cl', 'Br']
```

- To print one element per line we can loop over the inner list:

```
>>> elements = [[ 'Li', 'Na', 'K'], ['F', 'Cl', 'Br']]
>>> for inner_list in elements:
...     for item in inner_list:
...         print(item)
...
Li
Na
K
F
Cl
Br
```

Nested data structures are very common.

Looping Over Ragged Lists

- Nested lists don't have to be the same length
- Nested lists with inner lists of varying length are called *ragged lists*
- Python makes it easy to access ragged lists without knowing their length:

```
>>> info = [['Isaac Newton', 1643, 1727],
...     ['Charles Darwin', 1809, 1882],
...     ['Alan Turing', 1912, 1954, 'alan@bletchley.uk']]
>>> For record in info:
...     print(len(record), end='')
...     for field in record:
...         print('', field, end='')
...     print()
3 Isaac Newton 1643 1727
3 Charles Darwin 1809 1882
4 Alan Turing 1912 1954 alan@bletchley.uk
```

Ragged data structures can be hard to handle due to missing items.

Looping Until a Condition is Reached

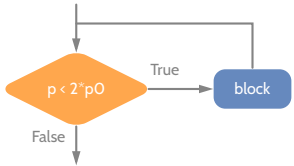
- To make **`for`** loops work Python needs to know when to stop iterating without the programmer's help
- There are many situations where this is not possible
- For example, when user input is involved: does the user want to quit or not?
- The general form of the **`while`** statement is:

```
while expression:
    block
```

- The expression is also called the *loop condition*
- The loop condition is checked; if it is **`True`** the block is executed
- This is repeated until the loop condition is **`False`**

We use `while` loops when the number of iterations depends on data items at run time.

Looping Until a Condition is Reached




```
def p_doubled(p0, r):
    t = 0
    p = p0
    while p < 2*p0:
        p += r * p
        print(round(p))
        t += 1

    print("P doubled in", t, "min.")
    print("Final P:", round(p))
```

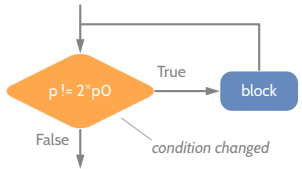
- Let's calculate the time it takes a bacterial colony to double its population
- With P being the population, r the growth rate per minute and t the time in minutes, we have:

$$P(t+1) = P(t) + rP(t)$$

```
>>> p_doubled(1000, 0.21)
1210
1464
1772
2144
P doubled in 4 min.
Final P: 2144
```

You can think of a `while` loop as a repeated `if` statement.

Infinite Loops



- Let's change the condition to require the population to be *exactly* doubled
- This does not work well, except for very few combinations of P_0 and r
- Python displays `inf` if the values are too large to be represented as a `float`

```
def p_doubled(p0, r):
    t = 0
    p = p0
    while p != 2*p0: # condition changed
        p += r * p
        print(round(p))
        t += 1

    print("P doubled in", t, "min.")
    print("Final P:", round(p))
```

```
>>> p_doubled(1000, 0.21)
1210
1464
1772
2144
2594
3138
*** several thousand lines later ***
inf
inf
*** and so on forever ***
```

Infinite loops are a common symptom of bugs.

Repetition Based on User Input

- We now can keep asking the user for input:

```
text = ""
while text != "quit":
    text = input("Please enter a chemical formula (or 'quit' to exit): ")
    if text == "quit":
        print("exiting program...")
    elif text == "H2O":
        print("Water")
    elif text == "NH3":
        print("Ammonia")
    elif text == "CH4":
        print("Methane")
    else:
        print("Unknown compound")
```

- The program will exit when the user types quit:

```
Please enter a chemical formula (or 'quit' to exit): H2O
Water
Please enter a chemical formula (or 'quit' to exit): quit
exiting program...
```

Note that the loop variable is defined before the loop header.

Controlling Loops Using Break & Continue

- As a rule, **`for`** and **`while`** loops execute the whole block on each iteration
- Sometimes it is useful to be able to break that rule
- Python provides two ways to control the loop from *within* the block:
  - **`break`** terminates the loop immediately
  - **`continue`** skips ahead to the next iteration
- Python also provides a loop **`else`** statement which is executed if and only if there was no **`break`** in the loop:

```
for expression:
    block
else:
    block (iff no break)
```

```
while expression:
    block
else:
    block (iff no break)
```

These all have their place. But use them with great care. Keep your code readable.

The Break Statement

- We demonstrate the **`break`** statement with compound formula program
- From the user's perspective the behaviour does not change
- The following is *not* an infinite loop:

```
while True:
    text = input("Please enter a chemical formula (or 'quit' to exit): ")
    if text == "quit":
        print("exiting program...")
        break
    elif text == "H2O":
        print("Water")
    elif text == "NH3":
        print("Ammonia")
    elif text == "CH4":
        print("Methane")
    else:
        print("Unknown compound")
```

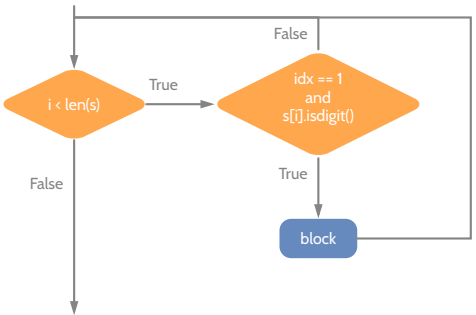
In situations like this explicit loop conditions are easier to read.

The Break Statement

- Sometimes a loop's task is finished early
- Without **`break`**, the loop has to finish iterating
- For example, we might want to know the index of the first digit in a string
- There is no need to continue iterating when we have found it

In this case `break` should be considered.

```
>>> s = 'C3H7'
>>> idx = -1
>>> for i in range(len(s)):
...     if idx == -1 and s[i].isdigit():
...         idx = i
...
>>> idx
1
```



The Break Statement

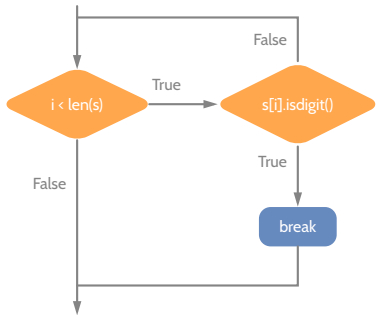
- We can simplify the code using **`break`**
- The **`if`** statement is much simpler now
- One can argue the code is more readable
- But what if there is no digit in the string?
- Or if the string is empty?
- That's a problem, because we omitted the definition

```
>>> idx = -1
```

- We could just add it back in, but there is another way…

It is a good thing if programs do less.

```
>>> s = 'C3H7'
>>> for i in range(len(s)):
...     if s[i].isdigit():
...         idx = i
...         break
...
>>> idx
1
```

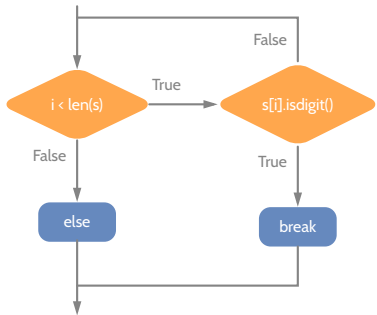


The Loop Else Statement

- Python provides the loop **`else`** statement
- It is executed when the loop finished iterating without interruption
- In particular, if there was no **`break`**
- It can be very useful to check for unexpected circumstances
- The readability can be debated

Some despise this syntax. We think it's OK.

```
>>> s = 'ABCDE'
>>> for i in range(len(s)):
...     if s[i].isdigit():
...         idx = i
...         break
... else:
...     print('Warning: no digit found.')
...     idx = -1
```

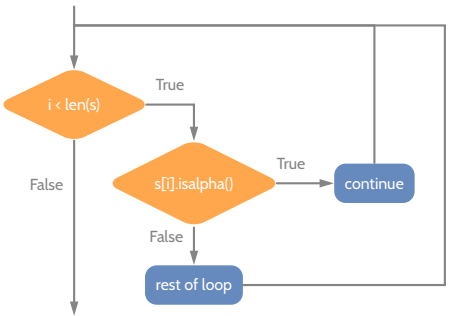


The Continue Statement

- We can also "bend the rules" with the **`continue`** statement
- For example, we might want to sum up all digits in a string
- We don't want to process letters
- One way of doing this is to skip letters with the **`continue`** statement

This works. But is it a good idea?

```
>>> s = 'C3H7'
>>> total = 0
>>> count = 0
>>> for i in range(len(s)):
...     if s[i].isalpha():
...         continue
...     total += int(s[i])
...     count += 1
...
>>> print(total, count)
10 2
```

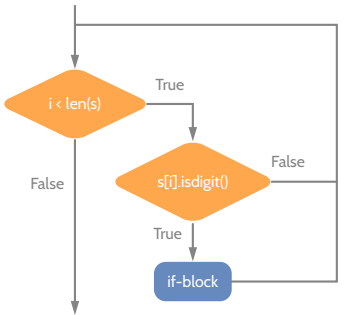


The Continue Statement

- We don't want to process letters
- That is, "if it is a digit, process it."
- If you say "if", write **`if`**
- It is much more readable here

Only use `continue` to avoid deep nesting

```
>>> s = 'C3H7'
>>> total = 0
>>> count = 0
>>> for i in range(len(s)):
...     if s[i].isdigit():
...         total += int(s[i])
...         count += 1
...
>>> print(total, count)
10 2
```



Exercises Lecture 9