# PROGRAMMING FUNDAMENTALS

## ITERATION

João Correia Lopes

INESC TEC, FEUP

09 October 2018

# GOALS

By the end of this class, the student should be able to:

- Describe how to do iterations using `while` statements
- Describe *middle-test* and *post-test* loops using the `break` and `continue` statements
- Choose between `for` and `while` loops
- Use nested loops for nested data (for example list of pairs)
- Trace a program
- Use Help and understand its meta-notation

# BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Section 3.3) [PDF]
- Brad Miller and David Ranum, Learning with Python: Interactive Edition. Based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers (Chapter 8) [HTML]

# TIPS

- This is a script and some illustrations used in the class, NOT a replacement for **reading the bibliography** listed in the *class sheet*
- “Students are responsible for anything that transpires during a class—therefore **if you’re not in a class**, you should get notes from someone else (not the instructor)”—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the class)
- We will be using **Moodle** as the primary means of communication

# CONTENTS

1 3.3 ITERATION

- 3.3.3 The for loop revisited
- 3.3.4 The while statement
- 3.3.5 The Collatz 3n + 1 sequence
- 3.3.6 Tracing a program
- 3.3.7 Counting digits
- 3.3.8 Help and meta-notation
- 3.3.9 Tables
- 3.3.10 Two-dimensional tables
- 3.3.11 The break statement
- 3.3.12 Other flavours of loops
- 3.3.13 An example
- 3.3.14 The continue statement
- 3.3.15 Paired Data
- 3.3.16 Nested Loops for Nested Data
- 3.3.17 Newton's method for finding square roots
- 3.3.18 Algorithms
- Exercises

- Computers are often used to automate repetitive tasks
- Repeating identical or similar tasks without making errors is something that computers do well and people do poorly
- Repeated execution of a set of statements is called **iteration**
- Python provides several language features to make it easier
  - We've already seen the `for` statement
  - We're going to look at the `while` statement
- Before we do that, let's just review a few ideas . . .

# ASSIGNMENT REVISITED

- It is legal to make more than one assignment to the same variable
- A new assignment makes an existing variable refer to a new value
- Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like `a = b` as a Boolean test.
- Unlike mathematics, it is not!
- Remember that the Python token for the equality operator is ==

⇒ https://github.com/fpro-admin/lectures/blob/master/05/assignment.py

# UPDATING VARIABLES REVISITED

- When an assignment statement is executed, the right-hand side expression (i.e. the *expression* that comes after the assignment token) is evaluated first
- This produces a *value*
- Then the assignment is made, so that the variable (assignable) on the left-hand side now *refers to* the new value

- Before you can update a variable, you have to **initialize** it to some starting value, usually with a simple assignment
- Updating a variable by adding 1 to it, is called an **increment**
- Subtracting 1 is called a **decrement**

⇒ `https://github.com/fpro-admin/lectures/blob/master/05/assignment.py`

# THE FOR LOOP REVISITED

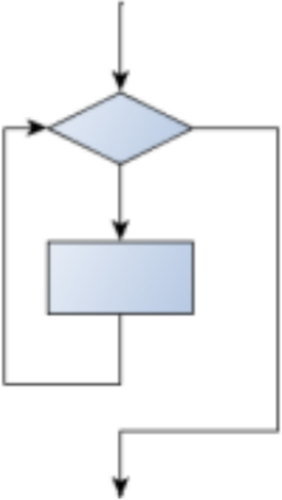
- Recall that the for loop processes each item in a list
- Each item in turn is (re-)assigned to the loop variable and the body of the loop is executed
- Running through all the items in a list is called **traversing the list**, or traversal

- *Let us write some code now to sum up all the elements in a list of numbers*

⇒ `https://github.com/fpro-admin/lectures/blob/master/05/for.py`

# THE while STATEMENT

```
while <CONDITION>:
    <STATEMENTS>
```



⇒ https://github.com/fpro-admin/lectures/blob/master/05/while.py

# INFINITE LOOP

- The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates
- Otherwise the loop will repeat forever, which is called an **infinite loop**

# CHOOSING BETWEEN FOR AND WHILE

- **Definite iteration** — we know ahead of time some definite bounds for what is needed
  - Use a for loop if you know, before you start looping, the maximum number of times that you'll need to execute the body
  - Examples: "iterate this weather model for 1000 cycles", or "search this list of words", "find all prime numbers up to 10000"
- **Indefinite iteration** — we're not sure how many iterations we'll need — we cannot even establish an upper bound!
  - if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (of if) this will happen, you'll need a while loop

# Collatz 3n + 1

## Computational rule

To create the sequence is to start from some given n, and
to generate the next term of the sequence from n,
either by halving n, (whenever n is even),
or else by multiplying it by three and adding 1.
The sequence terminates when n reaches 1.

```
n = 1027371
while n != 1:
    print(n, end=", ")
    if n % 2 == 0:
        n = n // 2
    else:
        n = n * 3 + 1
print(n, end=".\n")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/colatz.py

# Collatz 3n + 1

## Computational rule

To create the sequence is to start from some given n, and
to generate the next term of the sequence from n,
either by halving n, (whenever n is even),
or else by multiplying it by three and adding 1.
The sequence terminates when n reaches 1.

```
n = 1027371
while n != 1:
    print(n, end=", ")
    if n % 2 == 0:
        n = n // 2
    else:
        n = n * 3 + 1
print(n, end=".\n")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/colatz.py

# Tracing a program

- Tracing involves becoming the computer and following the *flow of execution* through a sample program run, recording the state of all *variables* and any *output* the program generates after each instruction is executed

| | |
|---|---|
| 3 | 3, |
| 10 | 3, 10, |
| 5 | 3, 10, 5, |
| 16 | 3, 10, 5, 16, |
| 8 | 3, 10, 5, 16, 8, |
| 4 | 3, 10, 5, 16, 8, 4, |
| 2 | 3, 10, 5, 16, 8, 4, 2, |
| 1 | 3, 10, 5, 16, 8, 4, 2, 1. |

# COUNTER

The following snippet counts the number of decimal digits in a positive integer:

```
n = 3029
count = 0
while n != 0:
   count = count + 1
   n = n // 10
print(count)
```

This snippet demonstrates an important pattern of computation called a **counter**.

⇒ https://github.com/fpro-admin/lectures/blob/master/05/counter.py

# Help and meta-notation

- Python comes with extensive documentation for all its built-in functions, and its libraries.
- See for example docs.python.org/3/library/...range
- The square brackets (in the description of the arguments) are examples of *meta-notation* — notation that describes Python syntax, but is not part of it
  - **range**([*start*,] *stop* [, *step*])
  - **for** *variable* **in** *list* **:**
  - **print**( [*object*, ... ] )
- Meta-notation gives us a concise and powerful way to describe the *pattern* of some syntax or feature.

# TABLE

- One of the things loops are good for is generating tables
- Output a sequence of values in the left column and 2 raised to the power of that value in the right column
    - using the “tab separator” *escape sequence*

```
    for x in range(11):  # Generate numbers 0 to 10
       print(x, "\t", 2**x)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/tables.py

# TWO-DIMENSIONAL TABLE

- A two-dimensional table is a table where you read the value at the intersection of a row and a column.
- Let's say you want to print a multiplication table for the values from 1 to 6
- A good way *to start* is to write a loop that prints the multiples of 2, all on one line:
    - `end="  "` argument in the print function suppresses the newline

```
    for i in range(1, 7):
        print(2 * i, end="   ")
    print()
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/tables.py

# THE BREAK STATEMENT

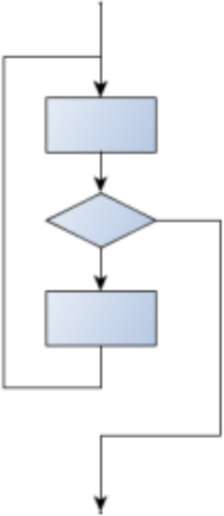
- The **break** statement is used to immediately leave the body of its loop
- The next statement to be executed is the first one after the body:

```
for i in [12, 16, 17, 24, 29]:
   if i % 2 == 1: # If the number is odd
      break       # ... immediately exit the loop
   print(i)
print("done.")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/break.py

# OTHER FLAVOURS OF LOOPS

- `for` and `while` loops do their tests at the start: They're called **pre-test loops**
- Sometimes we'd like to have the **middle-test loop** with the exit test in the middle of the body
- Or a **post-test loop** that puts its exit test as the last thing in the body



# MIDDLE-EXIT LOOP

```
total = 0
while True:
   response = input("Enter the next number. (Leave blank to end)")
      if response == "" or response == "-1":
         break
      total += int(response)
print("The total of the numbers you entered is ", total)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/break.py

# POST-TEST LOOP

```python
while True:
    play_the_game_once()
    response = input("Play again? (yes or no)")
    if response != "yes":
        break
print("Goodbye!")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/break.py

# A SIMPLE GUESSING GAME

- The *guessing game* program makes use of the mathematical law of **trichotomy**:
  - given real numbers a and b, exactly one of these three must be true:
  - a > b, a < b, or a == b

⇒ https://github.com/fpro-admin/lectures/blob/master/05/guess.py

# THE CONTINUE STATEMENT

- This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, for the current iteration
- But the loop still carries on running for its remaining iterations

```
 for i in [12, 16, 17, 24, 29, 30]:
     if i % 2 == 1:      # If the number is odd
         continue        # Don't process it
     print(i)
print("done.")
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/continue.py

# A pair of things

- Making a pair of things in Python is as simple as putting them into parentheses

```
# a pair
year_born = ("Kim Basinger", 1953)

# a list of pairs
celebs = [("Jack Nicholson", 1937), ("Kim Basinger", 1953),
             ("Brad Pitt", 1963), ("Sharon Stone", 1968)]
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/pairs.py

# Nested Loops for Nested Data

- Now we'll come up with an even more adventurous list of structured data
- In this case, we have a list of students
- Each student has a name which is paired up with another list of subjects that they are enrolled for

```
students = [
              ("John", ["FPRO", "LBAW"])
           ]
```

⇒ https://github.com/fpro-admin/lectures/blob/master/05/nested.py

# NEWTON'S METHOD

- Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it
- For example, in Newton's method for finding square root n.
- Starting with almost any approximation, a better approximation can be computed (closer to the actual answer) with the following formula:
  - `better = (approximation + n/approximation)/2`
- One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer (a great advantage for doing it manually)

⇒ `https://github.com/fpro-admin/lectures/blob/master/05/newton.py`

# ALGORITHMS

- Newton's method is an example of an **algorithm**:
    - it is a mechanical process for solving a category of problems (in this case, computing square roots)
- Some kinds of knowledge are not algorithmic:
    - learning dates from history or multiplication tables involves memorization of specific solutions
- But the techniques for addition with carrying, subtraction with borrowing, and long division are all algorithms
- One of the characteristics of algorithms is that they do not require any intelligence to carry out.
    - They are mechanical processes in which each step follows from the last according to a simple set of rules
- And they're designed to solve a general class or category of problems, not just a single problem

# ALGORITHMIC OR COMPUTATIONAL THINKING

### COMPUTATIONAL THINKING

Using algorithms and automation as the basis for approaching problems is rapidly transforming our society.

- Understanding that hard problems can be solved by step-by-step algorithmic processes (and having technology to execute these algorithms for us) is one of the major breakthroughs that has had enormous benefits
- But, some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically (e.g. NLP)

# EXERCISES

- Moodle activity at: LE05: Iteration