

# PROGRAMMING FUNDAMENTALS

## INTRODUCTION & THE WAY OF THE PROGRAM

João Correia Lopes

INESC TEC, FEUP

25 September 2018

# COMPUTER SCIENTISTS

- Like **mathematicians**, computer scientists use formal languages to denote ideas (specifically computations).
- Like **engineers**, they design things, assembling components into systems and evaluating tradeoffs among alternatives.
- Like **scientists**, they observe the behavior of complex systems, form hypotheses, and test predictions.

# PROBLEM SOLVING

The single most important skill for a computer scientist is **problem solving**.

- Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately.
- The process of learning to program is an excellent opportunity to practice problem-solving skills.

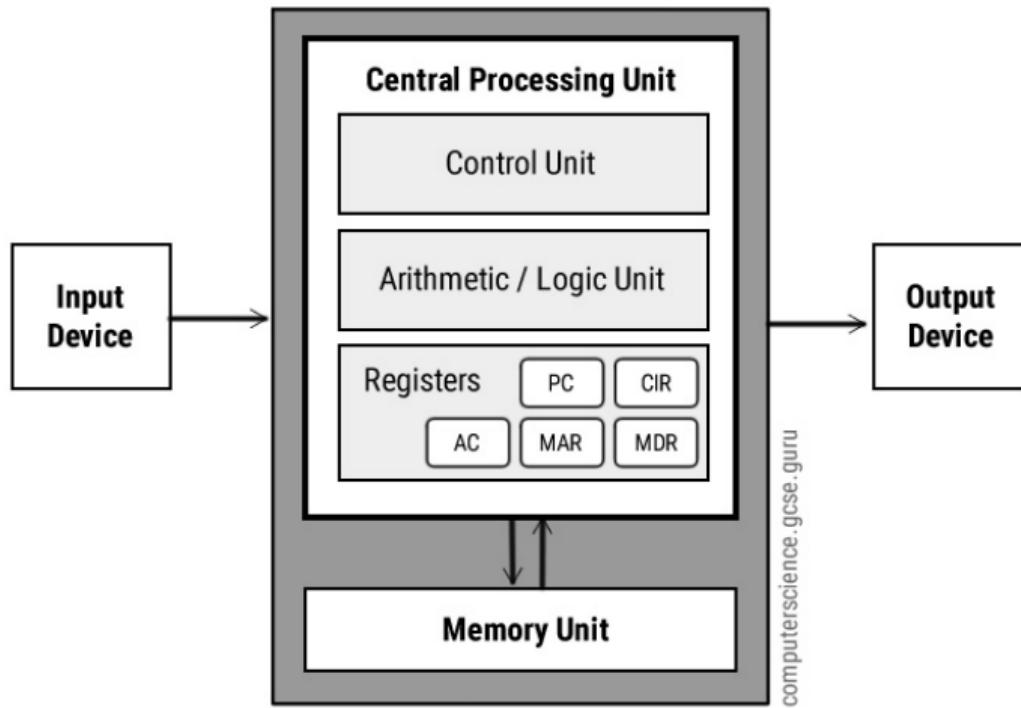
# ALGORITHM

- A set of specific steps for solving a category of problems

Soluções para os problemas

**steps + flow + stop decision**

# VON NEUMANN ARCHITECTURE



⇒ <https://www.computerscience.gcse.guru/theory/von-neumann-architecture>

# HIGH-LEVEL LANGUAGES

- Low-level languages (machine languages or assembly languages), are the only languages a computer executes;
- Thus, programs written in a high-level language have to be translated into something *more suitable* before they can run;
- Python is an example of a high-level language;
- Other high-level languages you might have heard of are C++, PHP, Pascal, C#, and Java.

# PYTHON INTERPRETER

There are two ways to use it:

- In *immediate mode*, you type Python expressions into the Python Interpreter window, and the interpreter immediately shows the result;
- In *script mode*, you can write a program in a file and use the interpreter to execute the contents of the file.

⇒ [Spyder3](#)

# TIME FOR KAHOOT!

⇒ <https://play.kahoot.it/#/?quizId=1f83b3ec-3aec-4fae-a52d-4a4a873bb1e5>

# PYTHON

```
1 #!/usr/bin/env python3\n\n3 import datetime\nnow = datetime.datetime.now()\n\n5\nprint()\n7 print("Current date and time using str method of datetime object:")\nprint()\n9 print(str(now))\n\n11 print()\nprint("Current date and time using instance attributes:")\n13 print()\nprint("Current year: %d" % now.year)\n15 print("Current month: %d" % now.month)\nprint("Current day: %d" % now.day)\n17 print("Current hour: %d" % now.hour)\nprint("Current minute: %d" % now.minute)\n19 print("Current second: %d" % now.second)\nprint("Current microsecond: %d" % now.microsecond)\n21\nprint()\n23 print("Current date and time using strftime:")\nprint(now.strftime("%Y-%m-%d %H:%M"))
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/01/basics.py>

## 1.1 THE PYTHON PROGRAMMING LANGUAGE

- Python is an interpreted **high-level** programming language for general-purpose programming
- Created by Guido van Rossum and first released in 1991,
- Python has a design philosophy that emphasizes code readability, notably using significant whitespace.
- It provides constructs that enable clear programming on both small and large scales
- Python features a dynamic type system and automatic memory management
- It supports multiple programming paradigms, including imperative, functional, procedural and object-oriented
- It has a large and comprehensive **standard library**

## 1.2 WHAT IS A PROGRAM?

- A program is a sequence of instructions that specifies how to perform a computation
- A few basic instructions appear in just about every language:

INPUT Get data from the keyboard, a file, or some other device (such as a sensor)

OUTPUT Display data on the screen or send data to a file or other device (such as a motor)

MATH Perform basic mathematical operations like addition and multiplication

CONDITIONAL EXECUTION Check for certain conditions and execute the appropriate sequence of statements

REPETITION Perform some action repeatedly, usually with some variation

## 1.2 WHAT IS A PROGRAM?

- A program is a sequence of instructions that specifies how to perform a computation
- A few basic instructions appear in just about every language:
  - INPUT** Get data from the keyboard, a file, or some other device (such as a sensor)
  - OUTPUT** Display data on the screen or send data to a file or other device (such as a motor)
  - MATH** Perform basic mathematical operations like addition and multiplication
  - CONDITIONAL EXECUTION** Check for certain conditions and execute the appropriate sequence of statements
  - REPETITION** Perform some action repeatedly, usually with some variation

# 1.3 WHAT IS DEBUGGING?

- Programming is a complex process, and because it is done by human beings, it often leads to errors
- Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

9/9

0800 Auton started ✓ { 1.270 9.027 877 025  
 1/000 - stoppt - auton ✓ { 1.270 9.027 877 025  
 13.26.03.01 MP - AC 1.270 9.027 877 025 (mark)  
 023 PRO > 2.130 976 05  
 comdr 2.130 976 05  
 Relays 622 in 023 failed special speed test  
 in future - new test.  
 Relays changed  
 1104 Started Cosine Tape (Sine check)  
 1525 Started Multi Adder Test.

1545 Relay #70 Panel F  
 (moth) in relay.

First actual case of bug being found.  
 1600 auton startd.  
 1700 closed down.

⇒ [Wikipedia](#)

## 1.4 SYNTAX ERRORS

- Syntax refers to the structure of a program and the rules about that structure
- For example, in English, a sentence must begin with a capital letter and end with a period
- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message

## 1.5 RUNTIME ERRORS

- A runtime error does not appear until you run the program.
- These errors are also called *exceptions* because they usually indicate that something exceptional (and bad) has happened.

## 1.6 SEMANTIC ERRORS

- With a semantic error in your program, it will run successfully, but it will not do the right thing
- The problem is that the program you wrote is not the program you wanted to write
- The meaning of the program (its *semantics*) is wrong

## 1.7 EXPERIMENTAL DEBUGGING

- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work (clues, inference, ...)
- Debugging is also like an experimental science

## 1.8 FORMAL AND NATURAL LANGUAGES

- **Natural languages** are the languages that people speak, such as **English**
- **Formal languages** are languages that are designed by people for specific applications
- For example, the **math notation** is a **formal language** that is particularly good at denoting relationships among numbers and symbols

***Programming languages are formal languages designed to express computations.***

- Syntax rules: *tokens* & structure
- *Parsing* a statement is needed to determine its structure

# 1.9 A TYPICAL FIRST PROGRAM (WITH BONUS)

```
# my first variable
2 greeting = "Hello"

4 #
# COMMENTING LINES #####
6 #
# to comment MANY lines at a time, highlight all of them then CTRL+1
8 # do CTRL+1 again to uncomment them;
# try it on the next two commented lines below!

10 whoami = "jlopes"
12 greeting = greeting + " " + whoami

14 # output the greeting
print("\n" + greeting + "!")
16 #
18 ##### AUTOCOMPLETE #####
19 #
# Spyder can autocomplete names for you.
# Start typing a variable name defined in your program and hit tab
22 # before you finish typing;
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/01/hello.py>

## 1.10 COMMENTS

- A comment in a computer program is text that is intended only for the human reader
- It is completely ignored by the interpreter

```
#-----  
2 # This demo program shows off how elegant Python is!  
# Written by John Snow, December 2015.  
4 #-----  
6 print("Hello, World!") # Isn't this easy!
```

## 1.10 COMMENTS

- A comment in a computer program is text that is intended only for the human reader
- It is completely ignored by the interpreter

```
#-----  
2 # This demo program shows off how elegant Python is!  
# Written by John Snow, December 2015.  
4 #-----  
6 print("Hello, World!") # Isn't this easy!
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/01/comments.py>

# EXERCISES

- Moodle activity at: [LE01: The way of the program](#)

# PROGRAMMING FUNDAMENTALS

## VARIABLES, EXPRESSIONS AND STATEMENTS

João Correia Lopes

INESC TEC, FEUP

27 September 2018

# CONTENTS

## 1 SIMPLE PYTHON DATA

- 2.1 Values and data types
- 2.2 Variables
- 2.3 Variable names and keywords
- 2.4 Statements
- 2.5 Evaluating expressions
- 2.6 Operators and operands
- 2.7 Type converter functions
- 2.8 Order of operations
- 2.9 Operations on strings
- 2.10 Input
- 2.11 Composition
- 2.12 The modulus operator

# PYTHON

```
1 #!/usr/bin/env python3
2
3 import datetime
4 now = datetime.datetime.now()
5
6 print()
7 print("Current date and time using str method of datetime object:")
8 print()
9 print(str(now))
10
11 print()
12 print("Current date and time using instance attributes:")
13 print()
14 print("Current year: %d" % now.year)
15 print("Current month: %d" % now.month)
16 print("Current day: %d" % now.day)
17 print("Current hour: %d" % now.hour)
18 print("Current minute: %d" % now.minute)
19 print("Current second: %d" % now.second)
20 print("Current microsecond: %d" % now.microsecond)
21
22 print()
23 print("Current date and time using strftime:")
24 print(now.strftime("%Y-%m-%d %H:%M"))
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/01/basics.py>

# VALUES AND DATA TYPES

- A **value** is one of the fundamental things that a program manipulates
- Values are classified into different classes, or **data types**
- **type()** is a function that tell us the **type of a value**

# VARIABLES

- A variable is a **name** that refers to a **value**
- The **assignment statement** (=) gives a value to a variable
- The assignment statement binds a *name*, on the left-hand side of the operator, to a *value*, on the right-hand side (**name = value**)
- Later, one can assign a different value to the same variable (this is different from maths!)
- The assignment token, =, should not be confused with the equals token, ==

⇒ [Visualise a state snapshot](#)

# VARIABLE NAMES

- **Variable names** can be arbitrarily long
- They can contain both letters and digits, but they have to begin with a letter or an underscore
- It is legal to use uppercase letters, but it is not done (by convention)
- Names should be “meaningful to the human readers” (not to be confused with “meaningful to the computer”)

# KEYWORDS

- Keywords define the language's syntax rules and structure
- They cannot be used as variable names

and	as	assert	break	class	continue
def	del	elif	else	except	exec
finally	for	from	global	if	import
in	is	lambda	nonlocal	not	or
pass	raise	return	try	while	with
yield	True	False	None		

# STATEMENTS

- A **statement** is an instruction that the Python interpreter can execute
- Statements don't produce any result
- Further to the assignment statement, there are others (`while` statements, `for` statements, `if` statements, `import` statements)

# EVALUATING EXPRESSIONS

- An **expression** is a combination of values, variables, operators, and calls to functions
- The Python interpreter evaluates expressions and displays its result (a value)
- A value all by itself is a simple expression, and so is a variable

# OPERATORS AND OPERANDS

- **Operators** are special tokens that represent computations like addition, multiplication and division
- The values the operator uses are called **operands**
- When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed
- Operations in Python (+, -, /) mean what they mean in mathematics
- Asterisk (\*) is the token for multiplication, and \*\* is the token for exponentiation

# TYPE CONVERTER FUNCTIONS

- Type converter functions `int()`, `float()` and `str()`
- will (attempt to) convert their arguments into types `int`, `float` and `str` respectively

# ORDER OF OPERATIONS

- When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**
- Python follows the same precedence rules for its mathematical operators that mathematics does (**PEMDAS**)
- Operators with the same precedence are evaluated from **left-to-right** (*left-associative*)
- An exception to the left-to-right left-associative rule is the exponentiation operator `**`

# OPERATIONS ON STRINGS

- One cannot perform mathematical operations **on strings**, even if the strings look like numbers
- The **+** operator represents **concatenation**, not addition
- The **\*** operator also works on strings; it performs **repetition**

# INPUT

- There is a built-in function in Python, `input()`, for getting input from the user
- The user of the program can enter the input and click OK
- The `input()` function always return a string (without the new-line)

# COMPOSITION

- One of the most useful features of programming languages is their ability to take small building blocks and **compose** them into larger chunks.
- Let's do four-step program that asks the user to input a value for the radius of a circle, and then computes the area of the circle from the formula  $\pi r^2$
- TIP: try to make code as simple as you can for the human to follow

⇒ <https://github.com/fpro-admin/lectures/blob/master/02/area.py>

# THE MODULUS OPERATOR

- The **modulus operator** works on integers (and integer expressions)
- and gives the **remainder** when the first number is divided by the second
- In Python, the modulus operator is a percent sign (%)
- It has the same precedence as the multiplication operator

⇒ <https://github.com/fpro-admin/lectures/blob/master/02/remainder.py>

# EXERCISES

- Moodle activity at: [LE02: Variables, expressions and assignments](#)

# PROGRAMMING FUNDAMENTALS

## PROGRAM FLOW WITH TURTLES

João Correia Lopes

INESC TEC, FEUP

02 October 2018

# CONTENTS

## 1 3.1 PROGRAM FLOW WITH TURTLES

- 3.1.1 Our first turtle program
- 3.1.2 Instances — a herd of turtles
- 3.1.3 The for loop
- 3.1.4 Flow of Execution of the for loop
- 3.1.5 The loop simplifies our turtle program
- 3.1.6 A few more turtle methods and tricks
- Exercises

# PYTHON MODULES

- There are many modules in Python that provide very powerful features that we can use in our own programs:
  - to do maths
  - to send email
  - to fetch web pages
  - ... and many others
- With `turtle` one creates turtles and get them to draw shapes and patterns
- ... but the aim is to develop the theme: “computational thinking”

# SIMPLE GRAPHICS

Every window contains a *canvas*, which is the area inside the window on which we can draw

```
1 import turtle           # Allows us to use turtles  
  
3 window = turtle.Screen()    # Creates a playground for turtles  
alex = turtle.Turtle()        # Create a turtle, assign to alex  
  
5 alex.forward(50)           # Tell alex to move forward by 50 units  
7 alex.left(90)              # Tell alex to turn by 90 degrees  
alex.forward(30)              # Complete the second side of a rectangle  
  
9 window.mainloop()          # Wait for user to close window
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/turtles.py>

## SIMPLE GRAPHICS (2)

```
1 import turtle
2
3 window = turtle.Screen()
4
5 window.bgcolor("lightgreen")      # Set the window background color
6 window.title("Hello, Tess!")      # Set the window title
7
8 tess = turtle.Turtle()
9 tess.color("blue")                # Tell tess to change her color
10 tess.pensize(3)                  # Tell tess to set her pen width
11
12 tess.forward(50)
13 tess.left(120)
14 tess.forward(50)
15
16 window.mainloop()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/turtles.py>

# INSTANCES

From a *class* (Turtle) one may have many *objects* (**instances**) of Turtle;  
Each instance has its own **state and behaviour**

```
1 import turtle
2
3 window = turtle.Screen()      # Set up the window and its attributes
4 window.bgcolor("lightgreen")
5 window.title("Tess & Alex")
6
7 tess = turtle.Turtle()        # Create tess and set some attributes
8 tess.color("hotpink");
9 tess.pensize(5)
10
11 alex = turtle.Turtle()       # Create alex
12 ...
```

# INSTANCES (2)

```
...
2 tess.forward(80)          # Make tess draw equilateral triangle
tess.left(120);
4 tess.forward(80);
tess.left(120);
6 tess.forward(80)
tess.left(120)            # Complete the triangle
8
tess.right(180)           # Turn tess around
10 tess.forward(80)         # Move her away from the origin

12 alex.forward(50)        # Make alex draw a square
alex.left(90)
14 alex.forward(50)
alex.left(90)
16 alex.forward(50)
alex.left(90)
18 alex.forward(50)
alex.left(90)
20
window.mainloop()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/herd.py>

# FOR LOOP

- It is a basic building block of all programs to be able to *repeat* some code, over and over again.
- In computer science, we refer to this repetitive idea as *iteration*
- it has a *loop variable*, an indented *loop body*, and a terminating condition

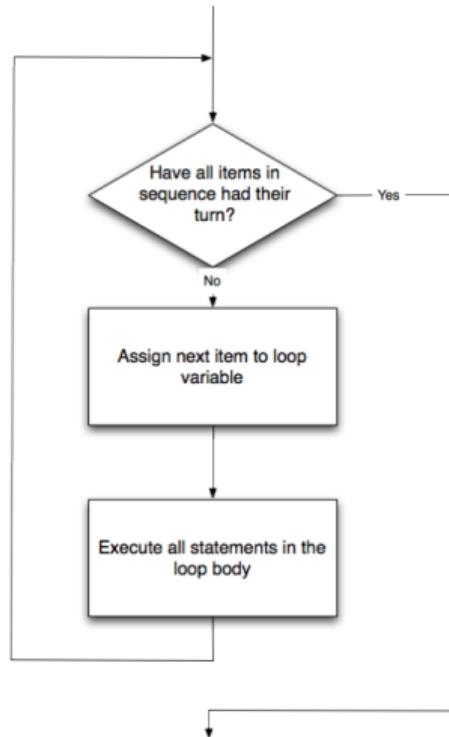
```
1 for friend in ["Joe", "Zoe", "Zuki", "Thandi", "Paris"]:  
    invite = "Hi " + friend + ". Please come to my party!"  
    print(invite)  
# more code to follow
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/for.py>

# FLOW OF EXECUTION

- As a program executes, the interpreter always keeps track of which statement is about to be executed.
- We call this the **control flow**, of the **flow of execution** of the program.
- Control flow until now has been strictly top to bottom, one statement at a time. The `for` loop changes this.
- See it in [pythontutor.com](http://pythontutor.com)

# FLOW OF EXECUTION OF THE FOR LOOP



# FOR LOOP EXAMPLE 1

```
1 import turtle           # set up alex
2 wn = turtle.Screen()
3 alex = turtle.Turtle()
4
5 for i in [0, 1, 2, 3]:  # repeat four times
6     alex.forward(50)
7     alex.left(90)
8
9 wn.exitonclick()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/for.py>

## FOR LOOP EXAMPLE 2

```
1 # 1
2 for aColor in ["yellow", "red", "purple", "blue"]:# repeat four times
3     alex.color(aColor)
4     alex.forward(50)
5     alex.left(90)
6
7
8 # 2
9 colors = ["yellow", "red", "purple", "blue"]
10 for color in colors:# for each color
11     alex.color(color)
12     alex.forward(50)
13     alex.left(90)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/for.py>

# THE RANGE FUNCTION

- Python gives us special built-in range objects
- Computer scientists like to count from 0!
- The most general form of the range is `range(start, beyondLast, step)`

```
1  for i in range(4):
2      # Executes the body with i = 0, then 1, then 2, then 3
3      print(i)
4
5  for _ in range(10):
6      # Sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
7      print(_)
8
9  for i in range(0, 20, 2):
10     print(i)
```

## SOME MORE TURTLE METHODS

- `tess.left(-30) / tess.right(330)` ?
- `alex.backward(-100) / alex.forward(100)` ?
- `alex.penup()` and `alex.pendown()`
- `alex.shape("turtle")`
- `alex.speed(10)`

# A FINAL EXAMPLE

```
1 import turtle
2
3 wn = turtle.Screen()
4 wn.bgcolor("lightgreen")
5
6 tess = turtle.Turtle()
7 tess.color("blue")
8 tess.shape("turtle")
9
10 print(list(range(5, 60, 2)))
11 tess.up()                      # this is new
12
13 for size in range(5, 60, 2):    # start with size = 5 and grow by 2
14     tess.stamp()                # leave an impression on the canvas
15     tess.forward(size)          # move tess along
16     tess.right(24)              # and turn her
17
18 wn.exitonclick()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/03/stamp.py>

# EXERCISES

- Moodle activity at: [LE03: Program Flow with Turtles](#)

# PROGRAMMING FUNDAMENTALS

## CONDITIONALS AND SELECTION

João Correia Lopes

INESC TEC, FEUP

04 October 2018

# CONTENTS

## 1 3.2 CONDITIONALS AND SELECTION

- 3.2.1 Boolean values and expressions
- 3.2.2 Logical operators
- 3.2.3 Truth Tables
- 3.2.4 Simplifying Boolean Expressions
- 3.2.5 Conditional execution
- 3.2.6 Omitting the else clause
- 3.2.7 Chained conditionals
- 3.2.8 Nested conditionals
- 3.2.9 Logical opposites
- Exercises

# BOOLEAN VALUES AND EXPRESSIONS

- Programs get really interesting when we can test conditions and change the program behaviour
- A *Boolean* value is either true or false
- In Python, the two Boolean values are True and False and the type is bool
- A Boolean expression is an expression that evaluates to produce a result which is a Boolean value
- For example, the operator == tests if two values are equal

# PYTHON

```
1 print(True)
2 print(type(True))
3 print(type(False))

5 print(type("True"))
#type(true)

7 print(5 == (3 + 2)) # 5 == 5 -> True
9
x = 5
11 print(x > 0 and x < 1) # 5 > 0 and 5 < 1 -> False

13 n = 25
14 print(n % 2 == 0 or n % 3 == 0) # 5%2 == 0 or 5%3 == 0 -> False
15
age = 19
17 old_enough_to_get_driving_licence = age >= 18 # 19 >= 18 -> True
18 print(old_enough_to_get_driving_licence)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/04/booleans.py>

# COMPARISON OPERATORS

```
x == y          # Produce True if ... x is equal to y  
2 x != y       # ... x is not equal to y  
x > y          # ... x is greater than y  
4 x < y          # ... x is less than y  
x >= y         # ... x is greater than or equal to y  
6 x <= y        # ... x is less than or equal to y
```

- There are three logical operators, `and`, `or`, and `not`
- to build more complex Boolean expressions from simpler Boolean expressions
- The semantics (meaning) of these operators is similar to their meaning in English
- The expression on the left of the `or` operator is evaluated first:
  - if the result is True, Python does not (and need not) evaluate the expression on the right
  - this is called *short-circuit evaluation*
- Similarly, for the `and` operator:
  - if the expression on the left yields False, Python does not evaluate the expression on the right

# TRUTH TABLE: AND

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

# TRUTH TABLE: OR

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

# TRUTH TABLE: NOT

a	not a
False	True
True	False

# PRECEDENCE OF OPERATORS

Level	Category	Operators
7(high)	exponent	**
6	multiplication	* , / , // , %
5	addition	+ , -
4	relational	== , != , <= , >= , > , <
3	logical	not
2	logical	and
1(low)	logical	or

# BOOLEAN ALGEBRA

- A set of rules for simplifying and rearranging expressions is called an *algebra*
- The *Boolean algebra* provides rules for working with Boolean values

# BOOLEAN ALGEBRA: AND

```
2     x and False == False
3     False and x == False
4     y and x == x and y
5     x and True == x
6     True and x == x
7     x and x == x
```

# BOOLEAN ALGEBRA: OR

```
2     x or False == x
3     False or x == x
4     y or x == x or y
5     x or True == True
6     True or x == True
7     x or x == x
```

# BOOLEAN ALGEBRA: NOT

```
not (not x) == x
```

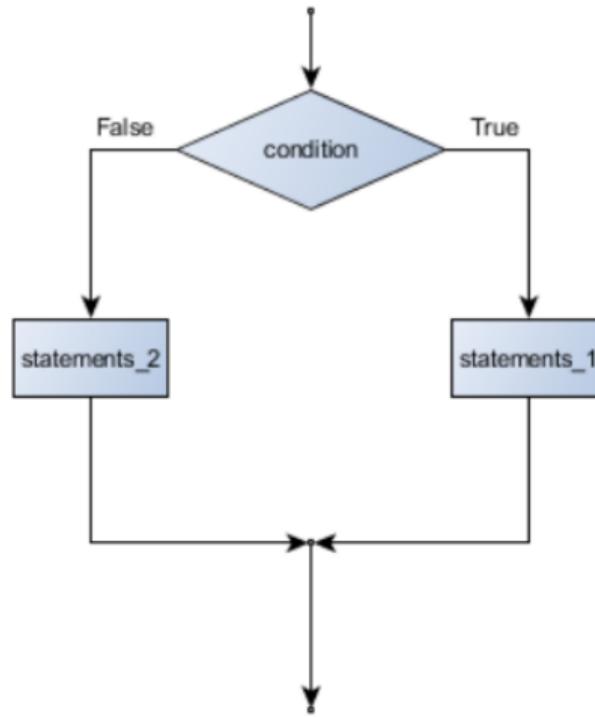
# CONDITIONAL STATEMENTS: **IF**

- Conditional statements give us the ability to check conditions and change the behavior of the program accordingly
- The simplest form is the *if statement*
- The Boolean expression after the if statement is called the condition

```
1     x = 15 #15 é ímpar logo irá avançar para o else  
2  
3     if x % 2 == 0:  
4         print(x, "is even") else:  
5             print(x, "is odd")
```

# IF STATEMENT WITH AN ELSE CLAUSE

```
2   if <BOOLEAN EXPRESSION>:  
3       <STATEMENTS_1>  
4   else:  
5       <STATEMENTS_2>
```



# BLOCKS AND INDENTATION

- The indented statements that follow are called a **block**
- The first unindented statement marks the end of the block
- There is no limit on the number of statements that can appear under the two clauses of an if statement
- but there has to be at least one statement in each block
- Occasionally, it is useful to have a section with no statements (usually as a place keeper, or scaffolding, for code we haven't written yet)

2

```
if True: # This is always True,  
    pass # so this is always executed, but it does nothing  
else:  
    pass # And this is never executed
```

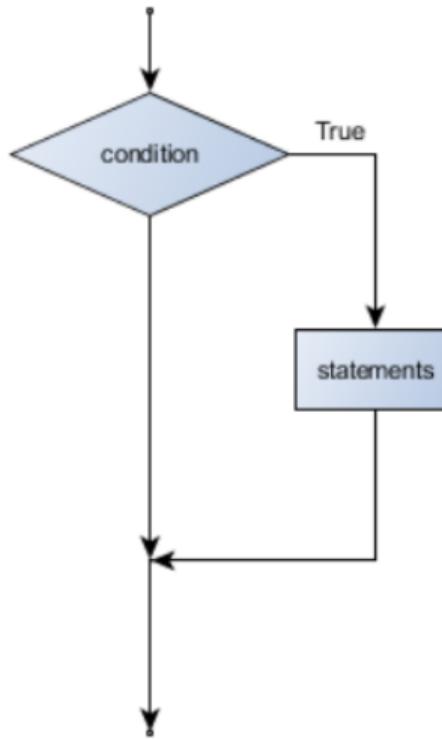
4

⇒ <https://github.com/fpro-admin/lectures/blob/master/04/selections.py>

# IF STATEMENT WITH NO ELSE CLAUSE: UNARY SELECTION

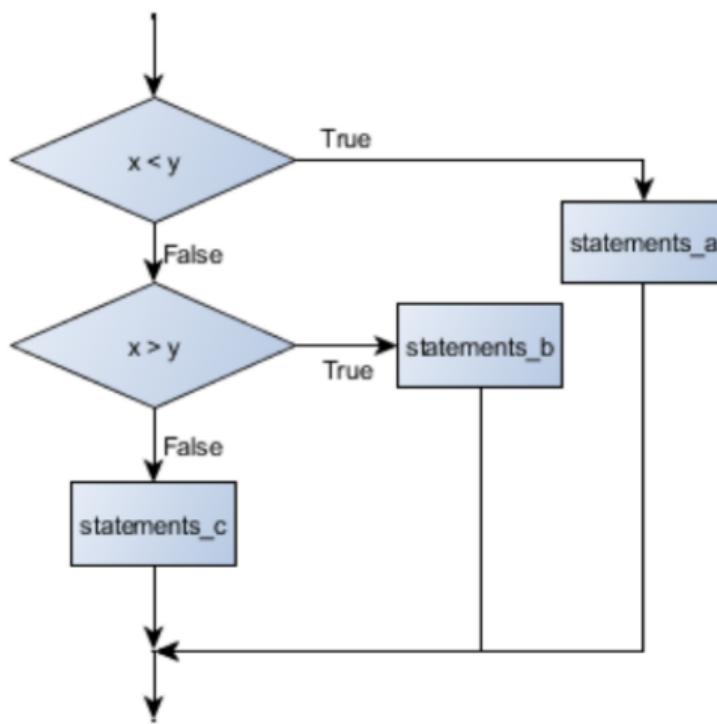
2

```
if <BOOLEAN EXPRESSION>:  
    <STATEMENTS>
```



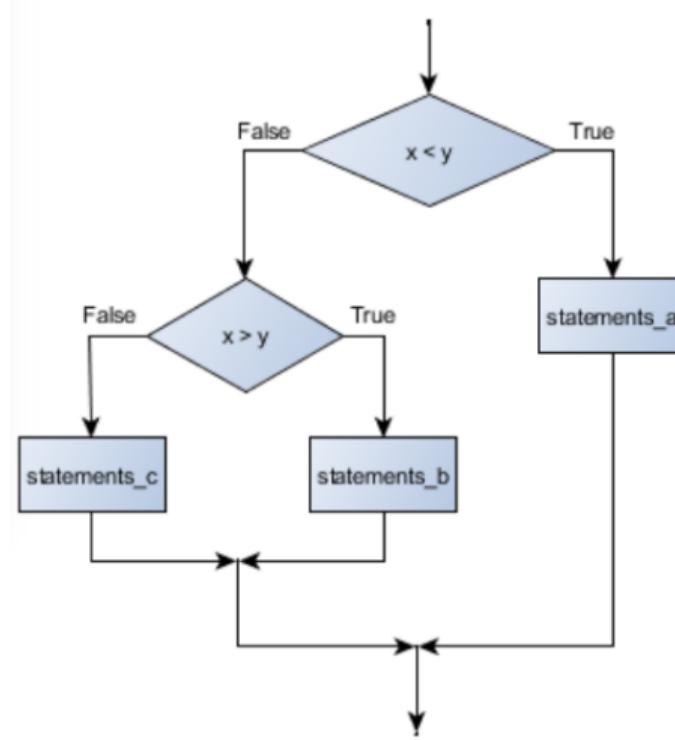
# CHAINED CONDITIONALS

```
2     if <BOOLEAN EXPRESSION_1>:  
3         <STATEMENTS_A>  
4     elif <BOOLEAN EXPRESSION_2>:  
5         <STATEMENTS_A>  
6     else:  
7         <STATEMENTS_C>
```



# NESTED CONDITIONALS

```
1 if <BOOLEAN EXPRESSION_1>:  
2     <STATEMENTS_A>  
3 else:  
4     if <BOOLEAN EXPRESSION_2>:  
5         <STATEMENTS_B>  
6     else:  
7         <STATEMENTS_C>
```



# LOGICAL OPPOSITES

operator	logical opposite
<code>==</code>	<code>!=</code>
<code>!=</code>	<code>==</code>
<code>&lt;</code>	<code>&gt;=</code>
<code>&lt;=</code>	<code>&gt;</code>
<code>&gt;</code>	<code>&lt;=</code>
<code>&gt;=</code>	<code>&lt;</code>

# EXERCISES

- Moodle activity at: [LE04: Conditionals](#)

# PROGRAMMING FUNDAMENTALS

## ITERATION

João Correia Lopes

INESC TEC, FEUP

09 October 2018

# 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>

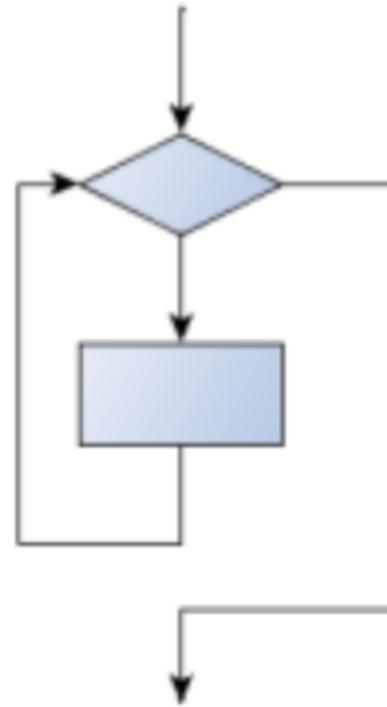
# THE FOR LOOP REVISITED

- 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

```
1 while <CONDITION>:  
2     <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<sup>1</sup> the condition becomes **false** and the **loop terminates**
- Otherwise the loop will repeat forever, which is called an **infinite loop**

---

<sup>1</sup>We use the adverb eventually to mean “in the end”, especially when something has involved a long time, or a lot of effort or problems [dictionary.cambridge.org]

# 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 (or if) this will happen, you'll need a **while** loop

# COLLATZ CONJECTURE

## COMPUTATIONAL RULE

The rule for creating 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.

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

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

# COLLATZ CONJECTURE

## COMPUTATIONAL RULE

The rule for creating 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.

```
1 n = 1027371
2 while n != 1: #a sequência termina quando n chegar a 1
3     print(n, end=", ")
4     if n % 2 == 0:
5         n = n // 2
6     else:
7         n = n * 3 + 1
8     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
- Let's try with the Collatz sequence of 3

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:

```
1  n = 3029
2  count = 0
3  while n != 0:          # quando count = 4, n = 0
4      count = count + 1  #0 -> 1; 1 -> 2; 2 -> 3; 3-> 4
5      n = n // 10        #3029 ->302; 302 -> 30; 30 -> 3; 3 -> 0
6  print(count)           # count = 4, o número tem 4 dígitos
```

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](https://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

```
1  print("n", '\t', "2**n")      #table column headings
2  print("----", '\t', "-----")
3
4  for x in range(11):          # generate values for columns
5      print(x, '\t', 2 ** x)
```

n	2**n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

⇒ <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

```
1  print("First line")
2  for i in range(1, 7):
3      print(2 * i, end="   ")
4      print()
5  print("\nTo be continued...")
```

2 4 6 8 10 12  
to be continued...

# 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*:

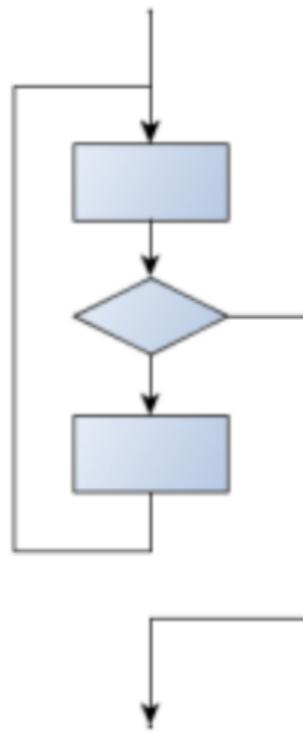
```
1  for i in [12, 16, 17, 24, 29]:  
2      if i % 2 == 1: # If the number is odd  
3          break      # ... immediately exit the loop  
4      print(i)  
5  print("done.")
```

12  
16  
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

```
Enter the next number (Leave blank to end): 2
Enter the next number (Leave blank to end): 3
Enter the next number (Leave blank to end): 4
Enter the next number (Leave blank to end):
The total of the numbers you entered is: 9
```

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

# POST-TEST LOOP

```
play_the_game_once()
```

```
Play again? (yes or no) yes  
play_the_game_once()
```

```
Play again? (yes or no) no  
Goodbye!
```

```
1 while True:  
2     play_the_game_once()  
3     response = input("Play again? (yes or no)")  
4     if response != "yes":  
5         break  
6     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

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

```
12  
16  
22  
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

```
1 # a pair
2 year_born = ("Kim Basinger", 1953)
3
4 # a list of pairs
5 celebs = [("Jack Nicholson", 1937),
6             ("Kim Basinger", 1953),
7             ("Brad Pitt", 1963),
8             ("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

```
1  students = [  
2      ("John", ["FPRO", "LBAW"] )  
3  ]
```

⇒ <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:
  - $\text{better} = (\text{approximation} + n/\text{approximation})/2$
- 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](#)

# PROGRAMMING FUNDAMENTALS

## MODULES, HELP, TRACE & TIPS

João Correia Lopes

INESC TEC, FEUP

11 October 2018

# CONTENTS

- 1 PYTHON MODULES
- 2 3.3.8 HELP & META-NOTATION
- 3 DEBUGGING INTERLUDE
- 4 3.3.6 TRACING A PROGRAM
- 5 3.4 TIPS, TRICKS & COMMON ERRORS
- 6 EXERCISES

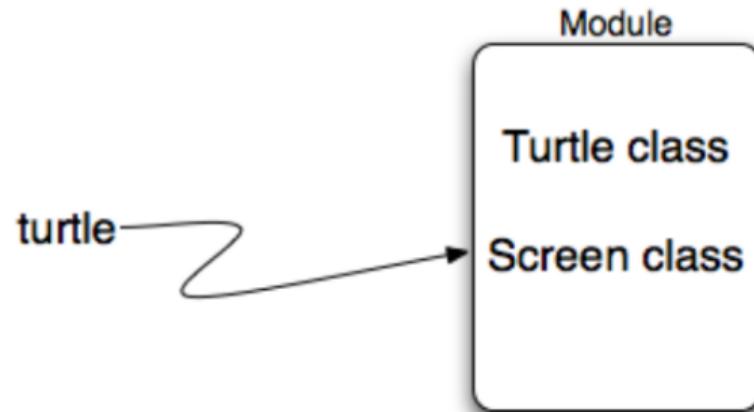
# PYTHON MODULES

- A **module** is a **file** containing Python **definitions and statements** intended for use in other Python programs
- There are many Python modules that come with Python as part of the **standard library**
- We have already used one of these quite extensively, the **turtle module**
- Recall that once we import the module, we can use things that are defined inside

```
⇒ https://docs.python.org/3.6/  
⇒ https://docs.python.org/3.6/library/  
⇒ https://docs.python.org/3/py-modindex.html  
⇒ https://docs.python.org/3.6/library/turtle.html  
⇒ https://github.com/python/cpython/blob/3.6/Lib/turtle.py
```

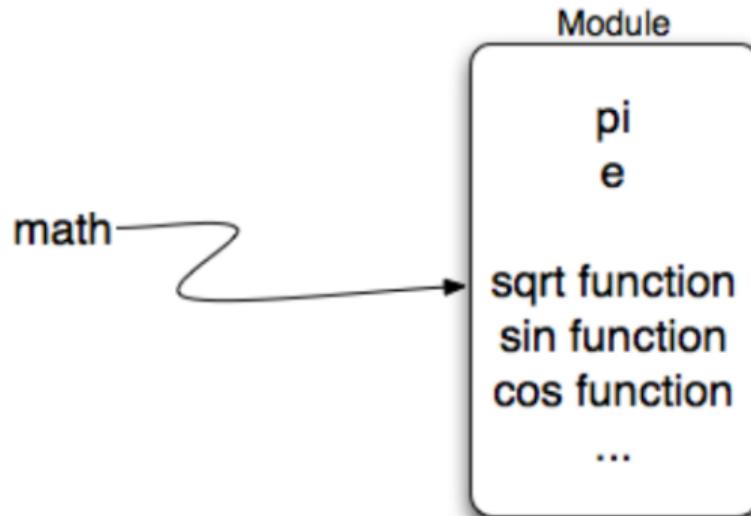
# USING MODULES

- The first thing we need to do when we wish to use a module is perform an import
- The statement `import turtle` creates a new name, `turtle`, and makes it refer to a module object
- This looks very much like the reference diagrams for simple variables



## THE MATH MODULE

- The math module contains the kinds of mathematical functions you would typically find on your calculator, and
- some mathematical constants like *pi* and *e*



⇒ <https://github.com/fpro-admin/lectures/blob/master/06/math.py>

# THE RANDOM MODULE

We often want to use **random numbers** in programs<sup>1</sup>.

Here are a few typical uses:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin
- To shuffle a deck of playing cards randomly
- To randomly allow a new enemy spaceship to appear and shoot at you
- To simulate possible rainfall when we make a computerised model for estimating the environmental impact of building a dam
- For encrypting your banking session on the Internet

⇒ <https://github.com/fpro-admin/lectures/blob/master/06/random.py>

---

<sup>1</sup>It is important to note that random number generators are based on a *deterministic algorithm* — repeatable and predictable. So they're called *pseudo-random generators* — they are not genuinely random.

## 3.3.8 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](https://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.

# HOW TO BE A SUCCESSFUL PROGRAMMER

- One of the most important skills you need to acquire is the ability to debug your programs
- Debugging is a skill that you need to master over time
- As programmers we spend 99% of our time trying to get our program to work
- But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good
- **Start small, get something small working, and then add to it**

# HOW TO AVOID DEBUGGING

## MANTRA

Get something working and keep it working

- **Start Small**

- This is probably the single biggest piece of advice for programmers at every level.

- **Keep it working**

- Once you have a small part of your program working the next step is to figure out something small to add to it.

Ok, let's look at an example: the `alarm_clock.py` of RE02

⇒ [https://github.com/fpro-admin/lectures/blob/master/06/alarm\\_clock.py](https://github.com/fpro-admin/lectures/blob/master/06/alarm_clock.py)

# BEGINNING TIPS FOR DEBUGGING

Debugging a program is a different way of thinking than writing a program.  
The process of debugging is much more like being a detective.

1 Everyone is a suspect (Except Python)!

2 Find clues

- Error messages
- Print statements

# BEGINNING TIPS FOR DEBUGGING

Debugging a program is a different way of thinking than writing a program.  
The process of debugging is much more like being a detective.

- 1 Everyone is a suspect (Except Python)!
- 2 Find clues
  - Error messages
  - Print statements

# SUMMARY ON DEBUGGING

- Make sure you take the time to understand error messages
  - They can help you a lot
- `print statements` are your friends
  - Use them to help you uncover what is **really** happening in your code
- Work backward from the error
  - Many times an error message is caused by something that has happened before it in the program
  - Always remember that python evaluates a program top to bottom

# 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
- Let's try with the Collatz sequence of 3

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.

# PROBLEMS WITH LOGIC AND FLOW OF CONTROL

We often want to know if some condition holds for any item in a list, e.g. “does the list have any odd numbers?”

This is a common mistake:

```
1     numbers = [10, 5, 24, 8, 6]
2     # Buggy version
3     for number in numbers:
4         if number % 2 == 1:
5             print(True)
6             break
7     else:
8         print(False)
9         break
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/06/odds.py>

## TIP: THINK ABOUT THE RETURN CONDITIONS OF THE LOOP

- Do I need to look at all elements in all cases?
- Can I shortcut and take an early exit?
- Under what conditions?
- When will I have to examine all the items in the list?

⇒ <https://github.com/fpro-admin/lectures/blob/master/06/odds.py>

# TIP: GENERALIZE YOUR USE OF BOOLEANS

- Programmers won't write `if is_prime(n) == True:` when they could say instead `if is_prime(n):`
- Think more generally about Boolean values, not just in the context of `if` or `while` statements.
- Like arithmetic expressions, they have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc
- See: [wikibooks](#)

## TIP: DON'T CREATE UNNECESSARY LISTS

- Lists are useful if you need to keep data for later computation
- But if you don't need lists, it is probably better not to generate them
- In the example below, there's two versions and both work
- What reasons are there for preferring the second version here?

⇒ <https://github.com/fpro-admin/lectures/blob/master/06/loops.py>

# EXERCISES

- Moodle activity at: [LE06: Modules, Help, tips & tricks](#)

# PROGRAMMING FUNDAMENTALS

## FUNCTIONS

João Correia Lopes

INESC TEC, FEUP

16 October 2018

# CONTENTS

- 1 4.1 FUNCTIONS
- 2 4.2 FUNCTIONS CAN CALL OTHER FUNCTIONS
- 3 4.3 FLOW OF EXECUTION
- 4 4.4 FUNCTIONS THAT REQUIRE ARGUMENTS
- 5 4.5 FUNCTIONS THAT RETURN VALUES
- 6 4.6 VARIABLES AND PARAMETERS ARE LOCAL
- 7 4.7 TURTLES REVISITED
- 8 EXERCISES

# FUNCTIONS

- A *function* is a **named sequence of statements that belong together**
- Their primary purpose is to **help us organize programs** into chunks that match how we think about the problem
- The syntax for a function definition is:

```
1 def <NAME>(<PARAMETERS>):  
2     <STATEMENTS>
```

- Function definitions are **compound statements**<sup>1</sup> which follow the pattern:
  - 1 A **header** line which begins with a **keyword** and ends with a **colon**
  - 2 A **body** consisting of *one or more* Python statements, each **indented** the same amount from the header line

---

<sup>1</sup>as was the case with `for` before

# DRAW A SQUARE

- Suppose we're working with turtles, and a common operation we need is to draw squares
- "Draw a square" is an abstraction, or a mental chunk, of a number of smaller steps
- So let's write a function to capture the pattern of this "building block"

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/turles.py>

# DOCSTRINGS FOR DOCUMENTATION

- If the first thing after the function header is a string, it is treated as a **docstring** and gets special treatment
- Docstrings are usually formed using triple-quoted strings
- Docstrings are the key way to document our functions in Python and the documentation part is important
- docstrings are not comments:
  - a string at the start of a function (a docstring) is retrievable by Python tools *at runtime*
  - comments are completely eliminated when the program is parsed

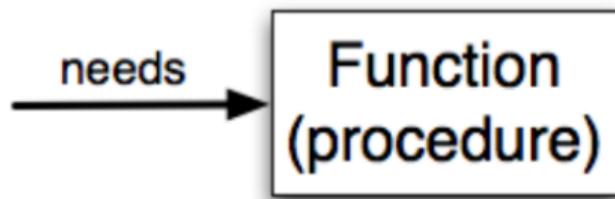
# FUNCTION CALL

- Defining the function just tells Python *how* to do a particular task, not to *perform* it
- In order to execute a function we need to make a **function call**
- Function calls contain the **name of the function** being executed followed by a **list of values**, called **arguments (actual parameters)**, which are assigned to the parameters in the function definition (**formal parameters**)
- Once we've defined a function, we can call it as often as we like, and **its statements will be executed each time we call it**

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/moreturles.py>

# ABSTRACTION

- The following diagram is often called a **black-box diagram** because it only states the requirements from the perspective of the user
- The user must know the name of the function and what arguments need to be passed
- The details of how the function works are hidden inside the “black-box”



# A SQUARE IS A (SPECIAL) RECTANGLE

- Let's assume now we want a function to draw a rectangle
- We may use it to draw a square

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/rectangle.py>

# FLOW OF EXECUTION

- Execution always begins at the first statement of the program
- Statements are executed one at a time, in order from top to bottom
- Function definitions do not alter the flow of execution of the program
  - statements inside the function are not executed until the function is called
- Function calls are like a detour in the flow of execution
  - Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/pytutor.py>

# TRACE A PROGRAM

## MORAL

When we read a program, don't read from top to bottom.  
Instead, follow the flow of execution.

# FUNCTIONS THAT REQUIRE ARGUMENTS

- Most functions require arguments: the arguments provide for generalisation

```
1  abs(5)          # 5
2  abs(-5)         # 5
3
4  pow(2, 3)        #8
5  pow(7, 4)        #2401
6
7  max(7, 11)       #11
8  max(4, 1, 17, 2, 12) #17
9  max(3 * 11, 5 ** 3, 512 - 9, 1024 ** 0) #503 (512 - 9)
```

# FUNCTIONS THAT RETURN VALUES

- A function that **returns a value** is called a **fruitful function**
- The **opposite of a fruitful function** is **void function** — one that is not executed for its resulting value (e.g. `draw_square`)
- Python will automatically return the value **`None`** for void functions (aka *procedures*)
- Most of the time, calling functions generates a value, which we usually assign to a variable or use as part of an expression

```
1 biggest = max(3, 7, 2, 5)
2 x = abs(3 - 11) + 10
```

# INTEREST RATES

The standard formula for compound interest:

$$A = P \left(1 + \frac{r}{n}\right)^{nt}$$

Where:

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = the number of times the interest is compounded per year
- t = the number of years that the interest is calculated for

Recall your implementation without functions. Cumbersome, right?

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/interests.py>

# VARIABLES AND PARAMETERS ARE LOCAL

- When we create a local variable inside a function, it only exists inside the function, and we cannot use it outside
- For example, consider again this function:

```
1 def final_amount(p, r, n, t):  
2     a = p * (1 + r/n) ** (n*t)  
3     return a
```

- If we try to use a, outside the function, we'll get an error
  - a only exists while the function is being executed — its lifetime
  - When the execution of the function terminates, the local variables are destroyed
  - Parameters are also local, and act like local variables
  - Remember, pythontutor is your friend!

# VARIABLES AND PARAMETERS ARE LOCAL

- When we create a local variable inside a function, it only exists inside the function, and we cannot use it outside
- For example, consider again this function:

```
1 def final_amount(p, r, n, t):  
2     a = p * (1 + r/n) ** (n*t)  
3     return a
```

- If we try to use `a`, outside the function, we'll get an error
- `a` only exists while the function is being executed — its **lifetime**
- When the **execution of the function terminates**, the **local variables are destroyed**
- Parameters are also local, and act like local variables
- Remember, `pythontutor` is your friend!

# VARIABLES AND PARAMETERS ARE LOCAL

- When we create a local variable inside a function, it only exists inside the function, and we cannot use it outside
- For example, consider again this function:

```
1 def final_amount(p, r, n, t):  
2     a = p * (1 + r/n) ** (n*t)  
3     return a
```

- If we try to use `a`, outside the function, we'll get an error
- `a` only exists while the function is being executed — its **lifetime**
- When the execution of the function terminates, the local variables are destroyed
- Parameters are also local, and act like local variables
- Remember, `pythontutor` is your friend!

# TURTLES REVISITED

- Now that we have fruitful functions, we can focus our attention on reorganizing our code so that it fits more nicely into our mental chunks
- This process of rearrangement is called **refactoring** the code

⇒ <https://github.com/fpro-admin/lectures/blob/master/07/refactoring.py>

# EXERCISES

- Moodle activity at: [LE07: Functions](#)

# PROGRAMMING FUNDAMENTALS

## FRUITFUL FUNCTIONS

João Correia Lopes

INESC TEC, FEUP

18 October 2018

# CONTENTS

- 1 4.10 RETURN VALUES
- 2 4.11 PROGRAM DEVELOPMENT
- 3 4.12 DEBUGGING WITH PRINT
- 4 4.13 COMPOSITION
- 5 4.14 BOOLEAN FUNCTIONS
- 6 4.15 PROGRAMMING WITH STYLE
- 7 EXERCISES

# RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1     biggest = max(3, 7, 2, 5)
```

```
1     x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

# RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1     biggest = max(3, 7, 2, 5)
```

```
1     x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

# RETURN VALUES

- The built-in functions we have used, such as `abs`, `pow`, `int`, `max`, and `range`, have produced results
- Calling each of these functions generates a value, which we usually assign to a variable or use as part of an expression

```
1     biggest = max(3, 7, 2, 5)
```

```
1     x = abs(3 - 11) + 10
```

- We are going to write more functions that return values, which we will call *fruitful functions*, for want of a better name.

# THE RETURN STATEMENT

- In a **fruitful function** the return statement includes a **return value**
- This statement means: evaluate the return expression, and then return it immediately as the result (the fruit) of this function
- Code that appears after a `return` statement<sup>1</sup> is called **dead code**

```
1 def area(radius):  
2     """returns the area of a circle with the given radius."""  
3     fruit = 3.14159 * radius ** 2  
4     return fruit
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/08/returns.py>

<sup>1</sup>or any other place the flow of execution can never reach

## MORE RETURNS

- All Python functions return `None` whenever they do not return another value
- It is also possible to use a `return` statement in the middle of a `for` loop, in which case control immediately returns from the function

⇒ <https://github.com/fpro-admin/lectures/blob/master/08/moreReturns.py>  
⇒ See it on [pythontutor](#)

# INCREMENTAL DEVELOPMENT

- To deal with increasingly complex programs, we are going to suggest a technique called incremental development
  - The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time
- 
- Suppose we want to find the *distance between two points*, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$
  - By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

⇒ <https://github.com/fpro-admin/lectures/blob/master/08/distance.py>

# INCREMENTAL DEVELOPMENT

- To deal with increasingly complex programs, we are going to suggest a technique called incremental development
- The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time
- Suppose we want to find the *distance between two points*, given by the coordinates  $(x_1, y_1)$  and  $(x_2, y_2)$
- By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

⇒ <https://github.com/fpro-admin/lectures/blob/master/08/distance.py>

## INCREMENTAL DEVELOPMENT (2)

The key aspects of the process are:

- 1 Start with a working skeleton program and make small incremental changes
- 2 Use temporary variables to refer to intermediate values so that you can easily inspect and check them
- 3 Once the program is working, relax, sit back, and play around with your options

### GOAL:

A good guideline is to aim for making code as easy as possible for others to read

## INCREMENTAL DEVELOPMENT (2)

The key aspects of the process are:

- 1 Start with a working skeleton program and make small incremental changes
- 2 Use temporary variables to refer to intermediate values so that you can easily inspect and check them
- 3 Once the program is working, relax, sit back, and play around with your options

### GOAL:

A good guideline is to aim for making code as easy as possible for others to read

# DEBUGGING WITH PRINT

- A powerful technique for debugging, is to insert **extra print functions** in carefully selected places in your code
- Then, by **inspecting the output of the program**, you can check whether the algorithm is doing what you expect it to
- Be clear about the following, however:
  - You must have a clear solution to the problem, and must know what should happen before you can debug a program
  - Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take
  - avoid calling `print` and `input` functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*<sup>2</sup>

---

<sup>2</sup>The exception is the `print` statements for debugging, later removed

# DEBUGGING WITH PRINT

- A powerful technique for debugging, is to insert extra `print` functions in carefully selected places in your code
- Then, by inspecting the output of the program, you can check whether the algorithm is doing what you expect it to
- Be clear about the following, however:
  - You must have a clear solution to the problem, and **must know what should happen** before you can debug a program
  - Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take
  - avoid calling `print` and `input` functions inside fruitful functions, *unless the primary purpose of your function is to perform input and output*<sup>2</sup>

---

<sup>2</sup>The exception is the `print` statements for debugging, later removed

# COMPOSITION

- **Composition** is the ability to call one function from within another
- As an example, we'll write a function that takes two points, the center of the circle ( $xc, yc$ ) and a point on the perimeter ( $xp, yp$ ), and computes the area of the circle

⇒ <https://github.com/fpro-admin/lectures/blob/master/08/area.py>

# BOOLEAN FUNCTIONS

- **Boolean functions** are functions that return Boolean values
  - which is often convenient for hiding complicated tests inside functions<sup>3</sup>

```
1  def is_divisible(x, y):  
2      """ Test if x is exactly divisible by y """  
3      if x % y == 0:  
4          return True  
5      else:  
6          return False
```

```
1  if is_divisible(x, y):  
2      ... # Do something ...  
3  else:  
4      ... # Do something else ...
```

---

<sup>3</sup>Can we avoid the if?

# BOOLEAN FUNCTIONS

- **Boolean functions** are functions that return Boolean values
  - which is often convenient for hiding complicated tests inside functions<sup>3</sup>

```
1  def is_divisible(x, y):
2      """ Test if x is exactly divisible by y """
3      if x % y == 0:
4          return True
5      else:
6          return False
```

```
1  if is_divisible(x, y):
2      ... # Do something ...
3  else:
4      ... # Do something else ..
```

---

<sup>3</sup>Can we avoid the if?

# PEP 8 — STYLE GUIDE FOR PYTHON CODE

- use 4 spaces (instead of tabs) for indentation
- limit line length to 78 characters
- when naming identifiers use `lowercase_with_underscores` for functions and variables
- place `imports` at the top of the file
- keep function definitions together below the `import statements`
- use `docstrings` to document functions
- use two blank lines to separate function definitions from each other
- keep top level statements, including function calls, together at the bottom of the program
- tip: Spyder3 may help you complying with PEP8...

# EXERCISES

- Moodle activity at: [LE08: Fruitful functions](#)

# PROGRAMMING FUNDAMENTALS

TESTING, MAIN, GLOBAL VARIABLES

João Correia Lopes

INESC TEC, FEUP

23 October 2018

# CONTENTS

- 1 UNIT TESTING
- 2 USING A MAIN FUNCTION
- 3 TRICKS & TIPS
- 4 GLOBAL VARIABLES
- 5 COMPUTATIONAL THINKING
- 6 EXERCISES

## 6.7. UNIT TESTING

- It is a common best practice in software development to include automatic unit testing of source code
- Unit testing provides a way to automatically verify that individual pieces of code, such as functions, are working properly
- This makes it possible to change the implementation of a function at a later time and quickly test that it still does what it was intended to do
- Unit testing also forces the programmer to think about the different cases that the function needs to handle
- Extra code in your program which is there because it makes debugging or testing easier is called **scaffolding**
- A collection of tests for some code is called its *test suite*

# UNIT TESTS

- At this stage we're going to ignore what the Python community usually does (see Codeboard), and we're going to code two simple functions ourselves
- Then we'll use these for writing our *unit tests*
- and we'll apply a *test suit* to `absolute_value()`
- First we plan our tests (think carefully about the “edge” cases):
  - 1 argument is negative
  - 2 argument is positive
  - 3 argument is zero

# HELPER FUNCTION

We're going to write a helper function for checking the results of one test.

```
1 import sys
2
3 def test(did_pass):
4     """ Print the result of a test. """
5     linenum = sys._getframe(1).f_lineno # the caller's line number
6     if did_pass:
7         msg = "Test at line {0} ok.".format(linenumber)
8     else:
9         msg = "Test at line {0} FAILED.".format(linenumber)
10    print(msg)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/tests.py>

# TEST SUIT

With the helper function written, we can proceed to construct our *test suite*:

```
1 def test_suite():
2     """
3         Run the suite of tests for code in this module (this file).
4     """
5     test(absolute_value(17) == 17)
6     test(absolute_value(-17) == 17)
7     test(absolute_value(0) == 0)
8     test(absolute_value(3.14) == 3.14)
9     test(absolute_value(-3.14) == 3.14)
10
11 test_suite()          # Here is the call to run the tests
```

See it with the `absolute_value(n)` buggy version.

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/tests.py>

# USING A MAIN FUNCTION

- Using functions is a good idea
- It helps us to modularize our code by breaking a program into logical parts where each part is responsible for a specific task
- For example, remember the function called `drawSquare()` that was responsible for having some turtle draw a square of some size
- The actual turtle and the actual size of the square were defined to be provided as parameters
- These final five statements of the program perform the main processing that the program will do

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/mainproc.py>

# PROGRAM STRUCTURE

- Now our program structure is as follows
  - 1 First, import any modules that will be required
  - 2 Second, define any functions that will be needed
  - 3 Third, define a main function that will get the process started
  - 4 And finally, invoke the main function (which will in turn call the other functions as needed)
- In Python there is nothing special about the name `main`<sup>1</sup>

---

<sup>1</sup>We could have called this function anything we wanted. We chose `main` just to be consistent with some of the other languages.

# ADVANCED TOPIC

- Before the Python interpreter executes your program, it defines a few special variables
- One of those variables is called `__name__`
- and it is automatically set to the string value "`__main__`" when the program is being executed by itself in a standalone fashion
- On the other hand, if the program is being imported by another program, then the "`__main__`" variable is set to the name of that module
- This ability to conditionally execute our main function can be extremely useful when we are writing code that will potentially be used by others
- For example, if we've a collection of functions to do some simple math...

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/mymath.py>  
⇒ <https://github.com/fpro-admin/lectures/blob/master/09/import.py>

# MAIN

```
1 #!/usr/bin/python3
2 # Filename: using_name.py
3
4 if __name__ == "__main__":
5     print("This program is being run by itself")
6 else:
7     print("I am being imported from another module")
```

# TIP: NONE IS NOT A STRING

- Values like `None`, `True` and `False` are not strings: they are special values in Python
- Keywords are special in the language: they are part of the syntax
- So we cannot create our own variable or function with a name `True` — we'll get a syntax error
- Built-in functions are not privileged like keywords: we can define our own variable or function called `len`, but we'd be silly to do so!

# TIP: UNDERSTAND WHAT THE FUNCTION NEEDS TO RETURN

- Perhaps functions return nothing
  - some functions exists purely to perform actions, rather than to calculate and return a result (*procedures*)
- But if the function should return a value
  - make sure all execution paths do return the value

# TIP: USE PARAMETERS TO GENERALIZE FUNCTIONS

- Understand which parts of the function will be hard-coded and unchangeable, and
- which parts should become parameters so that they can be customized by the caller of the function

# TIP: TRY TO RELATE PYTHON FUNCTIONS TO IDEAS WE ALREADY KNOW

- In math, we're familiar with functions like  $f(x) = 3x + 5$
- We already understand that when we call the function  $f(3)$  we make some association between the parameter  $x$  and the argument  $3$
- Try to draw parallels to argument passing in Python

# TIP: THINK ABOUT THE RETURN CONDITIONS OF THE FUNCTION

- Do I need to look at all elements in all cases?
- Can I shortcut and take an early exit?
- Under what conditions?
- When will I have to examine all the items in the list?

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/manyodd.py>

# TIP: GENERALIZE YOUR USE OF BOOLEANS

- Mature programmers won't write `if is_prime(n) == True:`
  - when they could say instead `if is_prime(n):`
- Like arithmetic expressions, booleans have their own set of operators (`and`, `or`, `not`) and values (`True`, `False`) and can be assigned to variables, put into lists, etc.

# LOCAL VARIABLES

- Tip: Local variables do not survive when you exit the function
- Tip: Assignment in a function creates a local variable

As soon as the function returns (whether from an explicit return statement or because Python reached the last statement), the stack frame and its local variables are all destroyed.

# GLOBAL VARIABLES

- UNLESS it makes use of variables that are **global**<sup>2</sup>

```
1     sz = 2
2     def h2():
3         """ Draw the next step of a spiral on each call. """
4         global sz
5         tess.turn(42)
6         tess.forward(sz)
7         sz += 1
```

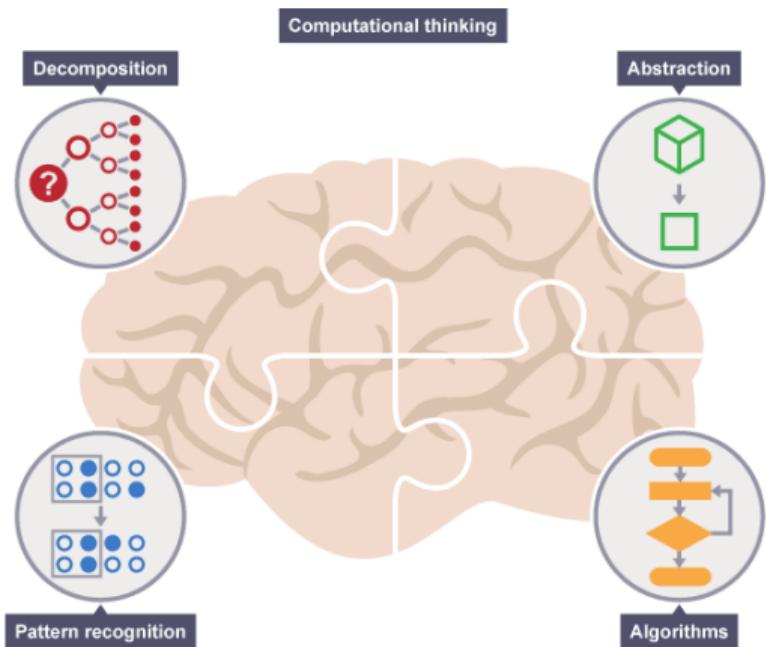
Each time we call `h2()` it turns, draws, and increases the global variable `sz`

---

<sup>2</sup>It's generally considered bad practice to use global variables. Functions should be as self-contained as possible (**no side-effects**).

# COMPUTATIONAL THINKING

Computational thinking allows us to take a complex problem, understand what the problem is and develop possible solutions. We can then present these solutions in a way that a computer, a human, or both, can understand.



# USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

DECOMPOSITION — breaking down a complex problem or system into smaller, more manageable parts

PATTERN RECOGNITION — looking for similarities among and within problems<sup>3</sup>

ABSTRACTION — focusing on the important information only, ignoring irrelevant details<sup>4</sup>

ALGORITHMS — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

→ BBC, Bitsize, [Introduction to computational thinking](#)

<sup>3</sup>Have any of the issues we've encountered in the past had solutions that could apply here?

<sup>4</sup>To make solutions as general as possible.

# USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

**DECOMPOSITION** — breaking down a complex problem or system into smaller, more manageable parts

**PATTERN RECOGNITION** — looking for similarities among and within problems<sup>3</sup>

**ABSTRACTION** — focusing on the important information only, ignoring irrelevant details<sup>4</sup>

**ALGORITHMS** — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

→ BBC, Bitsize, Introduction to computational thinking

<sup>3</sup>Have any of the issues we've encountered in the past had solutions that could apply here?

<sup>4</sup>To make solutions as general as possible.

# USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

**DECOMPOSITION** — breaking down a complex problem or system into smaller, more manageable parts

**PATTERN RECOGNITION** — looking for **similarities** among and within problems<sup>3</sup>

**ABSTRACTION** — focusing on the important information only, ignoring irrelevant details<sup>4</sup>

**ALGORITHMS** — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

→ BBC Bitsize, Introduction to computational thinking

<sup>3</sup>Have any of the issues we've encountered in the past had solutions that could apply here?

<sup>4</sup>To make solutions as general as possible.

# USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

**DECOMPOSITION** — breaking down a complex problem or system into smaller, more manageable parts

**PATTERN RECOGNITION** — looking for similarities among and within problems<sup>3</sup>

**ABSTRACTION** — focusing on the important information only, ignoring irrelevant details<sup>4</sup>

**ALGORITHMS** — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

→ BBC Bitsize, Introduction to computational thinking

<sup>3</sup>Have any of the issues we've encountered in the past had solutions that could apply here?

<sup>4</sup>To make solutions as general as possible.

## USE COMPUTATIONAL THINKING (2)

There are four key techniques (cornerstones) to computational thinking:

**DECOMPOSITION** — breaking down a complex problem or system into smaller, more manageable parts

**PATTERN RECOGNITION** — looking for similarities among and within problems<sup>3</sup>

**ABSTRACTION** — focusing on the important information only, ignoring irrelevant details<sup>4</sup>

**ALGORITHMS** — developing a step-by-step solution to the problem, or the rules to follow to solve the problem

⇒ [BBC, Bitsize, Introduction to computational thinking](#)

<sup>3</sup>Have any of the issues we've encountered in the past had solutions that could apply here?

<sup>4</sup>To make solutions as general as possible.

# EXERCISES

- Moodle activity at: [LE09: Testing, main and globals](#)

# PROGRAMMING FUNDAMENTALS

## DATA TYPES: STRINGS

João Correia Lopes

INESC TEC, FEUP

25 October 2018

# CONTENTS

## 1 DATA TYPES

- 5.1.1 A compound data type
- 5.1.2 Working with strings as single things
- 5.1.3 Working with the parts of a string
- 5.1.4 Length
- 5.1.5 Traversal and the `for` loop
- 5.1.6 Slices
- 5.1.7 String comparison
- 5.1.8 Strings are immutable
- 5.1.9 The `in` and `not in` operators
- 5.1.10 A `find` function
- 5.1.11 Looping and counting
- 5.1.12 Optional parameters
- 5.1.13 The built-in `find` method
- 5.1.14 The `split` method
- 5.1.15 Cleaning up your strings
- 5.1.16 The `string format` method
- Operations on Strings

# A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists and pairs
- Strings, lists, and pairs are qualitatively different from the others because they are made up of smaller pieces
- In the case of strings, they're made up of smaller strings each containing one **character** in a particular order from left to right
- Types that comprise smaller pieces are called **collection or compound data types**
- Depending on what we are doing, we may want to treat a compound data type as a single thing

# WORKING WITH STRINGS AS SINGLE THINGS

- Just like a turtle, a string is also an object
- So each string instance has its own attributes and methods (around 70!)
- For example:

```
1 >>> our_string = "Hello, World!"  
2 >>> all_caps = our_string.upper()  
3 >>> all_caps  
4 'HELLO, WORLD!'
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/smethods.py>

# WORKING WITH THE PARTS OF A STRING

- The **indexing operator** selects a single character substring from a string;

```
1  >>> fruit = "banana"      # a string
2  >>> letter = fruit[0]     # this is also a string
3  >>> print(letter)        # b
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/sindex.py>

# LENGTH

- The `len` function, when applied to a string, returns the number of characters in a string:

```
1 size = len(word)
2 last = word[size-1]
```

# TRAVERSAL AND THE FOR LOOP

- A lot of computations involve processing a *string one character at a time*
- Often they start at the beginning, select each character in turn, do something to it, and continue until the end
- This pattern of processing is called a traversal

```
1 word = "Banana"  
2 for letter in word:  
3     print(letter)
```

B  
a  
n  
a  
n  
a  
a

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/traversal.py>

# SLICES

- A **substring** of a string is obtained by **taking a slice**
- Similarly, we can slice a list to refer to some sublist of the items in the list

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/slices.py>

# STRING COMPARISON

- The comparison operators work on strings
- To see if two strings are equal:

```
1  if word == "banana":  
2      print("Yes, we have bananas!")
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/scomparison.py>

# STRINGS ARE IMMUTABLE

- Strings are **immutable**, which means you can't change an existing string
- The best you can do is **create a new string that is a variation on the original**

```
1 greeting = "Hello, world!"  
2 greeting[0] = 'J'           # ERROR!  
3  
4 greeting = "J" + greeting[1:]  
5 print(greeting)
```

# THE IN AND NOT IN OPERATORS

- The `in` operator tests for membership
- The `not in` operator returns the logical opposite results of `in`

```
1  >>> "p" in "apple"
2  True
3  >>> "i" in "apple"
4  False
5  >>> "apple" in "apple"
6  True
7  >>> "" in "a"
8  True
9  >>> "x" not in "apple"
10 True
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/removeVowels.py>

# A FIND FUNCTION

- In a sense, find is the opposite of the indexing operator
- What does the following function do?

```
1  def my_find(haystack, needle):
2      """
3          Find and return the index of needle in haystack.
4          Return -1 if needle does not occur in haystack.
5      """
6      for index, letter in enumerate(haystack):
7          if letter == needle:
8              return index
9      return -1
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/sfind.py>

# LOOPING AND COUNTING

- another example of the **counter** pattern introduced in *Counting digits*

```
1 def count_a(text):
2     count = 0
3     for letter in text:
4         if letter == "a":
5             count += 1
6     return count
7
8 print(count_a("banana") == 3)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/scount.py>

# OPTIONAL PARAMETERS

- To find the locations of the second or third occurrence of a character in a string
- we can modify the `find` function, adding a third parameter for the starting position in the search string
- Better still, we can combine `find` and `find2` using an **optional parameter**:

```
1  def find(haystack, needle, start=0):  
2      for index,letter in enumerate(haystack[start:]):  
3          if letter == needle:  
4              return index + start  
5      return -1
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/optional.py>

# THE BUILT-IN FIND METHOD

- The built-in `find` method is more general
- It can **find substrings**, not just single characters:

```
1  >>> "banana".find("nan")
2  2
3  >>> "banana".find("na", 3)
4  4
```

# THE SPLIT METHOD

- One of the most useful methods on strings is the **split method**
- it splits a single multi-word string into a list of individual words, removing all the whitespace between them<sup>1</sup>

```
1 >>> phrase = "Oh, that's jolly good. Well, off you go then"
2 >>> words = phrase.split()
3 >>> words
4 ['Oh,', "that's", 'jolly', 'good.', 'Well,', 'off', 'you', 'go', 'then']
```

<sup>1</sup>Whitespace means any tabs, newlines, or spaces.

# CLEANING UP YOUR STRINGS

- We'll often work with strings that contain punctuation, or tab and newline characters
- But if we're writing a program, say, to count word frequency, we'd prefer to strip off these unwanted characters.
- We'll show just one example of how to strip punctuation from a string
  - we need to traverse the original string and create a new string, omitting any punctuation

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/strip.py>

# THE STRING FORMAT METHOD

- The easiest and most powerful way to format a string in Python3 is to use the `format` method
- The template string contains place holders, ... {0} ... {1} ... {2} ... etc
- The `format` method substitutes its arguments into the place holders
- To see how this works, let's start with a few examples:

```
1 phrase = "His name is {0}!".format("Arthur")
2 print(phrase)
3
4 name = "Alice"
5 age = 10
6 phrase = "I am {0} and I am {1} years old.".format(age, name)
7 print(phrase)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/format.py>

# FORMAT SPECIFICATION

- Each of the replacement fields can also contain a **format specification**
- This modifies how the substitutions are made into the template, and can control things like:
  - whether the field is aligned to the left <, center ^, or right >
  - the width allocated to the field within the result string (a number like 10)
  - the type of conversion: we'll initially only force conversion to float, **f** or perhaps we'll ask integer numbers to be converted to hexadecimal using **x**)
  - if the type conversion is a float, you can also specify how many decimal places are wanted: typically, **.2f** is useful for working with currencies to two decimal places

⇒ <https://github.com/fpro-admin/lectures/blob/master/10/formatspec.py>

# COMMON STRING OPERATIONS

- See the Python Standard Library for a comprehensive list of operations on strings: [PSL](#)

# EXERCISES

- Moodle activity at: [LE10: Strings](#)

# PROGRAMMING FUNDAMENTALS

## DATA TYPES: TUPLES

João Correia Lopes

INESC TEC, FEUP

06 November 2018

# CONTENTS

## 1 DATA TYPES

- 5.1.1 A compound data type
- 5.2.1 Tuples are used for grouping data
- 5.2.2 Tuple assignment
- 5.2.3 Tuples as return values
- 5.2.4 Composability of Data Structures
- Operations on Tuples

# A COMPOUND DATA TYPE

- So far we have seen built-in types like int, float, bool, str and we've seen lists and pairs
- Strings, lists, and **tuples** are qualitatively different from the others because they are made up of smaller pieces
- Tuples group any number of items, of different types, into a single compound value
- Types that comprise smaller pieces are called **collection** or **compound data types**
- Depending on what we are doing, we may want to treat a compound data type as a single **thing**

# TUPLES ARE USED FOR GROUPING DATA

- A **data structure** is a mechanism for grouping and organizing data to make it easier to use
- We saw earlier that we could group together pairs of values:

```
1     >>> year_born = ("Paris Hilton", 1981)
```

- The pair is an example of a **tuple**
- Generalizing this, a tuple can be used to group any number of items into a single compound value

```
1     >>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
2
3     >>>
4     >>> empty_tuple=()
```

# OPERATIONS ON TUPLES

- A tuple lets us “chunk” together related information and use it as a single thing
- Tuples support the same sequence operations as strings
- The index operator selects an element from a tuple

⇒ <https://github.com/fpro-admin/lectures/blob/master/11/tmethods.py>

# TUPLE ASSIGNMENT

- Python has a very powerful tuple assignment feature
- Allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment

1

```
(name, surname, year_born, movie, year_movie, profession, birthplace) = julia
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/11/tassign.py>

# TUPLES AS RETURN VALUES

- Functions can always only return a single value
- By making that value a tuple, we can effectively group together as many values as we like, and return them together

⇒ [https://github.com/fpro-admin/lectures/blob/master/11/circle\\_stats.py](https://github.com/fpro-admin/lectures/blob/master/11/circle_stats.py)

# COMPOSABILITY OF DATA STRUCTURES

- Tuples items can themselves be other tuples
- Tuples may be **heterogeneous**, meaning that they can be composed of elements of different type

```
1 julia_more_info = ( ("Julia", "Roberts"), (8, "October", 1967),  
2                         "Actress", ("Atlanta", "Georgia"),  
3                         [ ("Duplicity", 2009),  
4                             ("Notting Hill", 1999),  
5                             ("Pretty Woman", 1990),  
6                             ("Erin Brockovich", 2000),  
7                             ("Eat Pray Love", 2010),  
8                             ("Mona Lisa Smile", 2003),  
9                             ("Oceans Twelve", 2004) ] )
```

# UPDATING TUPLES

- Tuples are immutable, which means you cannot update or change the values of tuple elements
- You are able to take portions of existing tuples to create new tuples

```
1  tup1 = (12, 34.56)
2
3  # Following action is not valid for tuples
4  # tup1[0] = 100
5
6  # So let's create a new tuple as follows
7  tup1 = (100,) + tup1[1:]
8  print(tup1)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/11/tmore.py>

# COMMON SEQUENCE OPERATIONS

- See the Python Standard Library for a comprehensive list of operations on tuples (and other sequences): [PSL](#)

# EXERCISES

- Moodle activity at: [LE11: Tuples](#)

# PROGRAMMING FUNDAMENTALS

## DATA TYPES: LISTS

João Correia Lopes

INESC TEC, FEUP

08 November 2018

# CONTENTS

## 1 DATA TYPES: LISTS

- 5.1.1 A compound data type
- 5.3.1 List values
- 5.3.2 Accessing elements
- 5.3.3 List length
- 5.3.4 List membership
- 5.3.5 List operations
- 5.3.6 List slices
- 5.3.7 Lists are mutable
- 5.3.8 List deletion
- 5.3.9 Objects and references
- 5.3.10 Aliasing
- 5.3.11 Cloning lists
- Using `zip()`

# A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists, pairs or tuples
- Strings, **lists**, and tuples are qualitatively different from the others because they are made up of smaller pieces
- Lists group any number of items, of different types, into a single compound value
- Types that comprise smaller pieces are called **collection** or **compound data types**
- Depending on what we are doing, we may want to treat a compound data type as a single thing

# LISTS

- A **list** is an **ordered collection of values**
- The values that make up a list are called its **elements**, or its **items**
- Lists and strings — and other collections that maintain the order of their **items** — are called **sequences**

# LIST VALUES

- There are several ways to create a new list

```
1 numbers = [10, 20, 30, 40]
2
3 words = ["spam", "bungee", "swallow"]
4
5 stuffs = ["hello", 2.0, 5, [10, 20]]
```

- A list within another list is said to be **nested**
- a list with no elements is called an **empty list**, and is denoted **[]**

# ACCESSING ELEMENTS

- The syntax for accessing the elements of a list is the index operator: []
  - the syntax is the same as the syntax for accessing the characters of a string
- The expression inside the brackets specifies the index
- Remember that the indices start at 0
- Negative numbers represent reverse indexing

```
1      >>> numbers = [10, 20, 30, 40]
2
3      >>> numbers[1]          #20
4      >>> numbers[-3]        #20
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/12/lindex.py>

# LIST LENGTH

- The function `len` returns the length of a list, which is equal to the number of its elements
- It is a good idea to use this value as the upper bound of a loop, as it accommodates changes in the list

```
1 horsemen = ["war", "famine", "pestilence", "death"]
2
3 for i in range(len(horsemen)):
4     print(horsemen[i])
5
6 len(["car makers", 1, ["Ford", "Toyota", "BMW"], [1, 2, 3]])
```

# LIST MEMBERSHIP

- `in` and `not in` are Boolean operators that **test membership** in a sequence

```
1  >>> horsemen = ["war", "famine", "pestilence", "death"]
2  >>> "pestilence" in horsemen
3  True
4  >>> "debauchery" in horsemen
5  False
6  >>> "debauchery" not in horsemen
7  True
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/12/students.py>

# LIST OPERATIONS

- The + operator **concatenates** lists:

```
1      >>> a = [1, 2, 3]
2      >>> b = [4, 5, 6]
3      >>> c = a + b
4      >>> c
5      [1, 2, 3, 4, 5, 6]
```

- Similarly, the \* operator **repeats a list** a given number of times:

```
1      >>> [0] * 4
2      [0, 0, 0, 0]
3      >>> [1, 2, 3] * 3
4      [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

# LIST SLICES

- The slice operations we saw previously with strings let us work with sublists:

```
1  >>> a_list = ["a", "b", "c", "d", "e", "f"]
2  >>> a_list[1:3]
3  ['b', 'c']
4
5  >>> a_list[:4]
6  ['a', 'b', 'c', 'd']
7
8  >>> a_list[3:]
9  ['d', 'e', 'f']
10
11 >>> a_list[:]
12 ['a', 'b', 'c', 'd', 'e', 'f']
```

# LISTS ARE MUTABLE

- Unlike strings, lists are mutable, which means we can change their elements
- An assignment to an element of a list is called item assignment

```
1     >>> fruit = ["banana", "apple", "quince"]
2     >>> fruit[0] = "pear"
3     >>> fruit[2] = "orange"
4     >>> fruit
5     ['pear', 'apple', 'orange']
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/12/lassign.py>

# LIST DELETION

- Using slices to delete list elements can be error-prone
- The `del` statement removes an element from a list

```
1      >>> a = ["one", "two", "three"]
2      >>> del a[1]
3      >>> a
4      ["one", "three"]
```

# OBJECTS AND REFERENCES

- Since strings are *immutable*, Python optimizes resources by making two names that **refer** to the same string value refer to the same object

```
1      >>> a = "banana"
2      >>> b = "banana"
3      >>> a is b
4      True
5
6      >>> a = [1, 2, 3]
7      >>> b = [1, 2, 3]
8      >>> a == b
9      True
10
11     >>> a is b
12     False
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/12/references.py>

# ALIASING

- Since variables refer to objects, if we assign one variable to another, both variables refer to the same object
- Although this behavior can be useful, it is sometimes unexpected or undesirable

```
1      >>> a = [1, 2, 3]
2      >>> b = a
3
4      >>> a is b
5      True
6
7      >>> b[0] = 5
8      >>> a
9      [5, 2, 3]
```

# CLONING LISTS

- If we want to **modify a list** and also **keep a copy of the original**
- The easiest way to **clone** a list is to use the **slice operator**

```
1      >>> a = [1, 2, 3]
2      >>> b = a[:]      # considered bad practice
3      >>> c = a.copy() # better in Python3
4
5      >>> b
6      [1, 2, 3]
7
8      >>> b[0] = 5
9
10     >>> a
11     [1, 2, 3]
```

# USING ZIP()

```
1 coordinate = ['x', 'y', 'z']
2 value = [3, 4, 5]
3
4 result = zip(coordinate, value)
5 resultList = list(result)
6 print(resultList)
7
8 c, v = zip(*resultList)
9 print("c =", c)
10 print("v =", v)
```

[('x', 3), ('y', 4), ('z', 5)]  
c = ('x', 'y', 'z')  
v = (3, 4, 5)

⇒ <https://github.com/fpro-admin/lectures/blob/master/12/zip.py>  
⇒ <http://www.pythontutor.com/visualize.html>

# LIST OPERATIONS

- See the Python Standard Library for a comprehensive list of “Common Sequence Operations”: [PSL](#)
- See the Python Standard Library for a comprehensive list of operations on “Mutable Sequence Types”: [PSL](#)

# EXERCISES

- Moodle activity at: [LE12: Lists](#)

# PROGRAMMING FUNDAMENTALS

## DATA TYPES: WORKING WITH LISTS

João Correia Lopes

INESC TEC, FEUP

13 November 2018

# CONTENTS

## 1 DATA TYPES: LISTS

- 5.1.1 A compound data type
- 5.3.12 Lists and `for` loops
- 5.3.13 List parameters
- 5.3.14 List methods
- 5.3.15 Pure functions and modifiers (make side-effects)
- 5.3.16 Functions that produce lists
- 5.3.17 Strings and lists
- 5.3.18 Type conversions: `list` and `range`
- 5.3.19 Looping and lists
- 5.3.20 Nested lists
- 5.3.21 Matrices

# A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and we've seen lists, pairs or tuples
- Strings, **lists**, and tuples are qualitatively different from the others because they are made up of smaller pieces
- Lists (and tuples) group any number of items, of different types, into a single compound value
- Types that comprise smaller pieces are called **collection** or **compound data types**
- Depending on what we are doing, we may want to treat a compound data type as a single thing

# LISTS AND FOR LOOPS

- The `for` loop also works with lists, as we've already seen
- The generalized syntax of a `for` loop is:

```
1  for <VARIABLE> in <LIST>:  
2      <BODY>
```

- “For (every) friend in (the list of) friends, print (the name of the) friend”

```
1  friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]  
2  for friend in friends:  
3      print(friend)
```

# LIST EXPRESSIONS IN FOR LOOPS

- Any list expression can be used in a `for` loop
- `enumerate` generates pairs of both `(index, value)` during the list traversal

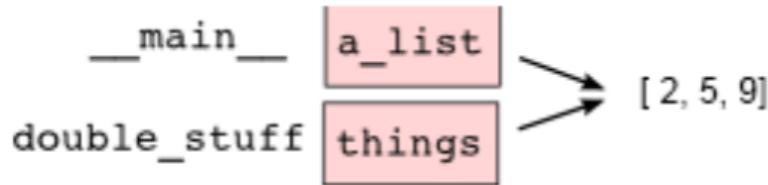
```
1  xs = [1, 2, 3, 4, 5]
2
3  for (i, val) in enumerate(xs):
4      xs[i] = val**2
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/13/for-lists.py>

## LIST PARAMETERS

- Passing a list as an argument actually passes a *reference* to the list, **not a copy** or clone of the list
- So parameter passing creates an alias<sup>1</sup>

```
1 def double_stuff(things):
2     """ Overwrite each element in a_list with double its value. """
3     ...
4     a_list = [2, 5, 9]
5     double_stuff(a_list)
```



<sup>1</sup>The caller has one variable referencing the list, and the called function has an alias, but there is only one underlying list object

# LIST METHODS

- The dot operator can also be used to access built-in methods of list objects

```
1      >>> mylist = []
2      >>> mylist.append(5)          # Add 5 onto the end of mylist
3      >>> mylist.append(12)
4      >>> mylist
5      [5, 12]
6
7      >>> mylist.insert(1, 12)    # Insert 12 at pos 1, shift others
8      >>> mylist.count(12)       # How many times is 12 in mylist?
9
10     >>> mylist.extend([5, 9, 5, 11])
11     >>> mylist.index(9)        # Find index of first 9 in mylist
12     >>> mylist.reverse()
13     >>> mylist.sort()
14     >>> mylist.remove(12)       # Remove the first 12 in mylist
```

# PURE FUNCTIONS AND MODIFIERS

- As seen before, there is a difference between a pure function and one with side-effects
- Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**
- A **pure function** does not produce side effects
  - It communicates with the calling program only through parameters, which it does not modify, and a return value

⇒ [https://github.com/fpro-admin/lectures/blob/master/13/double\\_stuff.py](https://github.com/fpro-admin/lectures/blob/master/13/double_stuff.py)

# FUNCTIONS THAT PRODUCE LISTS

- Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
1 initialize a result variable to be an empty list
2 loop
3     create a new element
4     append it to result
5 return the result
```

```
1 def primes_lessthan(n):
2     """ Return a list of all prime numbers less than n. """
3     result = []
4     for i in range(2, n):
5         if is_prime(i):
6             result.append(i)
7     return result
```

# FUNCTIONS THAT PRODUCE LISTS

- Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
1 initialize a result variable to be an empty list
2 loop
3     create a new element
4     append it to result
5 return the result
```

```
1 def primes_lessthan(n):
2     """ Return a list of all prime numbers less than n. """
3     result = []
4     for i in range(2, n):
5         if is_prime(i):
6             result.append(i)
7     return result
```

# STRINGS AND LISTS

- Two of the most useful methods on strings involve conversion to and from lists of substrings
- The `split` method breaks a string into a list of words
- By default, any number of whitespace characters is considered a word boundary
- An optional argument called a **delimiter** can be used to specify which string to use as the boundary marker between substrings
- The inverse of the `split` method is `join`
- You choose a desired **separator** string and `join` the list with the glue between each of the elements

⇒ <https://github.com/fpro-admin/lectures/blob/master/13/strings.py>

# L I S T

- Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list

```
1  >>> letters = list("Crunchy Frog")
2  >>> letters
3  ["C", "r", "u", "n", "c", "h", "y", " ", "F", "r", "o", "g"]
4  >>> "".join(letters)
5  'Crunchy Frog'
```

## RANGE

- One particular feature of `range` is that it doesn't instantly compute all its values:
  - it "puts off" the computation, and does it on demand, or "lazily"
  - We'll say that it gives a **promise** to produce the values when they are needed
  - This is very convenient if your computation short-circuits a search and returns early

⇒ <https://github.com/fpro-admin/lectures/blob/master/13/lazy-eval.py>

## LIST AND RANGE

- You'll sometimes find the lazy range wrapped in a call to list
- This forces Python to turn the lazy promise into an actual list

```
1      >>> range(10)          # Create a lazy promise
2      range(0, 10)
3
4      >>> list(range(10))    # Call in the promise, to produce a list
5      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# LOOPING AND LISTS

- Computers are useful because they can repeat computation, accurately and fast
- So loops are going to be a central feature of almost all programs you encounter

## TIP: DON'T CREATE UNNECESSARY LISTS

Lists are useful if you need to keep data for later computation.  
But if you don't need lists, it is probably better not to generate them.

⇒ <https://github.com/fpro-admin/lectures/blob/master/13/sums.py>

# LOOPING AND LISTS

- Computers are useful because they can repeat computation, accurately and fast
- So loops are going to be a central feature of almost all programs you encounter

## TIP: DON'T CREATE UNNECESSARY LISTS

Lists are useful if you need to keep data for later computation.  
But if you don't need lists, it is probably better not to generate them.

# NESTED LISTS

- A nested list is a list that appears as an element in another list

```
1      >>> nested = ["hello", 2.0, 5, [10, 20]]  
2      >>> print(nested[3])  
3      [10, 20]  
4  
5      >>> nested[3][1]  
6      20
```

# MATRICES

- Nested lists are often used to represent **matrices**<sup>2</sup>
- For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1      >>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
2  
3      >>> mx[1]          # select a row  
4      [4, 5, 6]  
5  
6      >>> mx[1][2]      # extract a single element  
7      6
```

<sup>2</sup>Later we will see a more radical alternative using a dictionary.

# MATRICES

- Nested lists are often used to represent matrices<sup>2</sup>
- For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
1  >>> mx = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
2  
3  >>> mx[1]          # select a row  
4  [4, 5, 6]  
5  
6  >>> mx[1][2]      # extract a single element  
7  6
```

<sup>2</sup>Later we will see a more radical alternative using a dictionary.

# EXERCISES

- Moodle activity at: [LE13: Working with Lists](#)

# PROGRAMMING FUNDAMENTALS

## DATA TYPES: DICTIONARIES

João Correia Lopes

INESC TEC, FEUP

15 November 2018

# CONTENTS

## 1 DATA TYPES: DICTIONARIES

- 5.1.1 A compound data type
- 5.4.1 Dictionary operations
- 5.4.2 Dictionary methods
- 5.4.3 Aliasing and copying
- 5.4.4 Counting letters
- 20.4 Sparse matrices
- 20.5 Memoization

# A COMPOUND DATA TYPE

- So far we have seen built-in types like `int`, `float`, `bool`, `str` and also lists, pairs or tuples
- Strings, lists, and tuples are qualitatively different from the others because they are made up of smaller pieces
- Lists, tuples, and strings have been called *sequences*, because their items occur in order
- Dictionaries group any number of items, of different types, into a single compound value
- Dictionaries are not sequences!

# DICTIONARY

- Dictionaries are yet another kind of compound type
- They are Python's built-in **mapping type**
- They map **keys**, which can be any immutable type, to **values**, which can be any type (heterogeneous)<sup>1</sup>
- In other languages, they are called **associative arrays** since they associate a key with a value
- One way to create a dictionary is to start with the **empty dictionary** and **add key:value pairs**

```
1      >>> english_spanish = {}  
2      >>> english_spanish['one'] = "uno"  
3      >>> english_spanish["two"] = 'dos'  
4      >>> print(english_spanish)  
5      {'one': 'uno', 'two': 'dos'}
```

---

<sup>1</sup>Just like the elements of a list or tuple

# HASHING

- The order of the pairs may not be what was expected
- Python uses complex algorithms, designed for very fast access, to determine where the key:value pairs are stored in a dictionary
- For our purposes we can think of this ordering as **unpredictable**
- The implementation uses a technique called **hashing**
- The same concept of mapping a key to a value could be implemented using a list of tuples, but...

```
1  >>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
2  {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}
3
4  >>> [("apples", 430), ("bananas", 312), ("oranges", 525), ("pears", 217)]
5  [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

# LOOK UP A VALUE



⇒ <https://en.wikipedia.org/wiki/Mafalda>

## LOOK UP A VALUE

- Another way to create a dictionary is to provide a list of **key:value** pairs using the same syntax as the previous output
- It doesn't matter what order we write the pairs (there's no indexing!)<sup>2</sup>

```
1  >>> english_spanish = {"one": "uno", "three": "tres", "two": "dos"}  
2  >>> english_spanish  
3  {'one': 'uno', 'three': 'tres', 'two': 'dos'}  
4  
5  >>> print(english_spanish["two"])  
6  dos
```

<sup>2</sup>The dictionary is the first compound type that we've seen that is not a sequence, so we can't index or slice a dictionary

# DICTIONARY OPERATIONS

- The `del` statement removes a `key:value` pair from a dictionary
- The `len` function also works on dictionaries; it returns the number of `key:value` pairs

```
1     >>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
2     >>> del inventory["bananas"]
3     >>> len(inventory)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/14/operations.py>

# DICTIONARY METHODS

- Dictionaries have a number of useful built-in methods
- The `keys()` method returns what Python3 calls a view of its underlying keys
  - A view object has some similarities to the range object we saw earlier — it is a lazy promise, to deliver its elements when they're needed by the rest of the program
  - We can iterate over the view, or turn the view into a list like this
- The `values()` method is similar
- The `items()` method also returns a view, which promises a list of tuples

```
1  for key in english_spanish.keys(): # The order of the k's is not defined
2      print("Got key", key, "which maps to value", english_spanish[key])
```

# ALIASING AND COPYING

- As in the case of lists, because **dictionaries are mutable**, we need to be aware of aliasing
- Whenever two variables refer to the same object, changes to one affect the other
- If we want to modify a dictionary and keep a copy of the original, use the **copy** method

```
1  >>> opposites = {"up": "down", "right": "wrong", "yes": "no"}  
2  >>> alias = opposites  
3  >>> copy = opposites.copy()  # Shallow copy
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/14/methods.py>

# GENERATE A FREQUENCY TABLE

- To write a function that counted the number of occurrences of a letter in a string
- Dictionaries provide an elegant way to generate a frequency table

```
1     start with an empty dictionary
2     for each letter in the string:
3         find the current count (possibly zero) and increment it
4     the dictionary contains pairs of letters and their frequencies
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/14/frequency-table.py>

# SPARSE MATRICES

- We previously used a list of lists to represent a matrix
- That is a good choice for a matrix with mostly nonzero values, but consider a sparse matrix like this one:

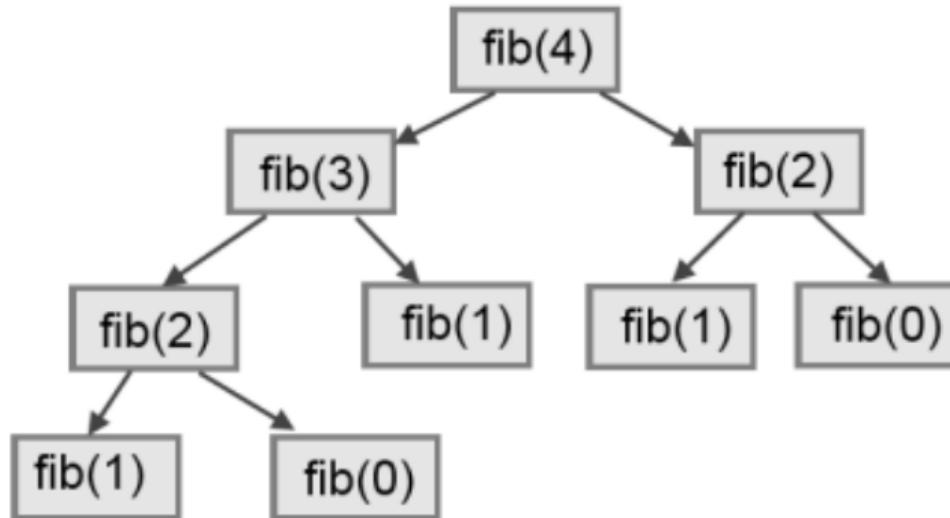
$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

- The list representation contains a lot of zeroes
- An alternative is to use a dictionary and the `get()` method
- There's a trade-off here as the access may take more time

⇒ <https://github.com/fpro-admin/lectures/blob/master/14/matrix.py>

# MEMOISATION

- Consider this call graph for `fib()` with  $n = 4$
- A good solution is to keep track of values that have already been computed by storing them in a dictionary
- A previously computed value that is stored for later use is called a **memo**



⇒ <https://github.com/fpro-admin/lectures/blob/master/14/fib.py>

# EXERCISES

- Moodle activity at: [LE14: Dictionaries](#)

# PROGRAMMING FUNDAMENTALS

LIST ALGORITHMS

João Correia Lopes

INESC TEC, FEUP

20 November 2018

# CONTENTS

## 1 ALGORITHMS THAT WORK WITH LISTS

- The linear search algorithm [14.2]
- A more realistic problem [14.3]
- Binary Search [14.4]
- Removing adjacent duplicates from a list [14.5]
- Merging sorted lists [14.6]
- Alice in Wonderland, again! [14.7]
- Summary

# LIST ALGORITHMS

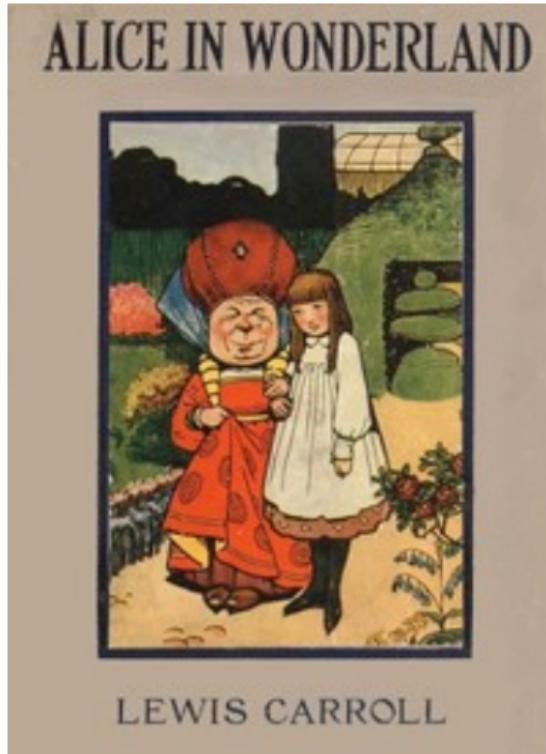
MPFC:

**And now for something completely different!**

- Rather than introduce more programming constructs, or new Python syntax and features
- ... we focus on some algorithms that work with lists

# ALICE IN WONDERLAND

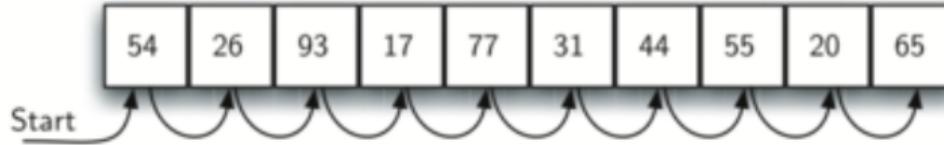
- Examples work with the book “Alice in Wonderland” and a “vocabulary file”



⇒ [Alice in Wonderland](#)  
⇒ [Vocabulary](#)

# THE LINEAR SEARCH ALGORITHM

- We'd like to know the index where a specific item occurs within in a list of items
- Specifically, we'll return the index of the item if it is found, or we'll return -1 if the item doesn't occur in the list



⇒ [InteractivePython](#)

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/linear.py>

# LINEAR SEARCH

```
1  def search_linear(xs, target):
2      """ Find and return the index of target in sequence xs """
3      for (i, v) in enumerate(xs):
4          if v == target:
5              return i
6      return -1
```

- Searching all items of a sequence from first to last is called a **linear search**
- Each time we check whether `v == target` we'll call it a **probe**
  - We like to count probes as a measure of how efficient our algorithm is
  - this will be a good enough indication of how long our algorithm will take to execute
- Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on the length of the list
- On average, when the target is present, we're going to need to go about halfway through the list, or  $N/2$  probes

# LINEAR SEARCH

```
1  def search_linear(xs, target):
2      """ Find and return the index of target in sequence xs """
3      for (i, v) in enumerate(xs):
4          if v == target:
5              return i
6      return -1
```

- Searching all items of a sequence from first to last is called a **linear search**
- Each time we check whether `v == target` we'll call it a **probe**
  - We like to count probes as a measure of how efficient our algorithm is
  - this will be a good enough indication of how long our algorithm will take to execute
- Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on the length of the list
- On average, when the target is present, we're going to need to go about halfway through the list, or  $N/2$  probes

# LINEAR PERFORMANCE

- We say that linear search has **linear performance** (linear meaning *straight line*)
- Analysis like this is pretty **meaningless for small lists**
  - The computer is quick enough not to bother if the list only has a handful of items
- So generally, we're interested in the **scalability** of our algorithms
  - How do they perform if we throw **bigger problems** at them
  - What happens for really large datasets, e.g. how does Google search so brilliantly well?

# A MORE REALISTIC PROBLEM

- As children learn to read, there are expectations that their vocabulary will grow
- So a child of age 14 is expected to know more words than a child of age 8
- When prescribing reading books for a grade, an important question might be “**which words in this book are not in the expected vocabulary at this level?**”
  - Let us assume we can read a vocabulary of words into our program
  - Then read the text of a book, and split it into words

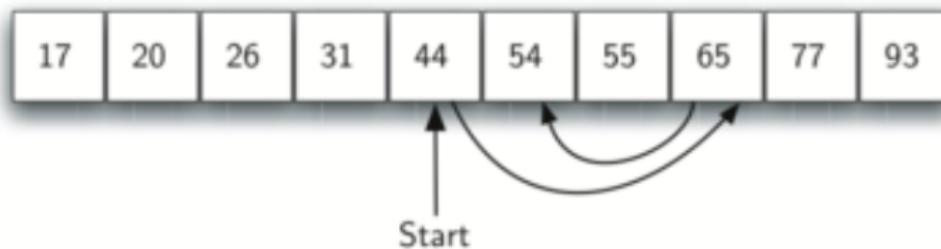
⇒ <https://github.com/fpro-admin/lectures/blob/master/15/alice.py>

# BINARY SEARCH

- If you were given a vocabulary and asked to tell if some word was present, you'd probably start in the middle
- You can do this because the **vocabulary is ordered** — so you can probe some word in the middle, and immediately realize that your target was before (or perhaps after) the one you had probed
- Applying this principle repeatedly leads us to a very much better algorithm for searching in a list of items that are already ordered
- This algorithm is called **binary search**
- It is a good example of *divide and conquer*

# REGION OF INTEREST (ROI)

- Our algorithm will start with the ROI set to all the items in the list
- On the first probe in the middle of the ROI, there are three possible outcomes:
  - either we **find the target**
  - or we learn that we can **discard the top half** of the ROI
  - or we learn that we can **discard the bottom half** of the ROI
- Trying with 54...



⇒ [InteractivePython](#)

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/binary.py>

## BACK TO “A MORE REALISTIC PROBLEM”

- What a spectacular difference! More than 200 times faster!
- If we uncomment the print statement on lines 15 and 16, we'll get a trace of the probes done during a search

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/alice.py>

# REMOVING ADJACENT DUPLICATES FROM A LIST

- We often want to get the unique elements in a list, i.e. produce a new list in which each different element occurs just once
- Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary
- We had a report that there are 3398 such words, but there are duplicates in that list
- In fact, the word “alice” occurs 398 times in the book, and it is not in our vocabulary!
- How should we remove these duplicates?
- A good approach is to sort the list, then remove all adjacent duplicates

# REMOVING ADJACENT DUPLICATES FROM A LIST

- We often want to get the unique elements in a list, i.e. produce a new list in which each different element occurs just once
- Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary
- We had a report that there are 3398 such words, but there are duplicates in that list
- In fact, the word “alice” occurs 398 times in the book, and it is not in our vocabulary!
- How should we remove these duplicates?
- A good approach is to sort the list, then **remove all adjacent duplicates**

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/adjacent.py>

## BACK TO “A MORE REALISTIC PROBLEM”

- Let us go back now to our analysis of *Alice in Wonderland*
- Before checking the words in the book against the vocabulary, we'll sort those words into order, and eliminate duplicates
- Lewis Carroll was able to write a classic piece of literature using only 2570 different words!

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/alice2.py>

# MERGING SORTED LISTS

- Suppose we have two sorted lists (`xs` and `ys`)
- Devise an algorithm to merge them together into a single sorted list
- A simple but inefficient algorithm could be to simply append the two lists together, and sort the result
- But this doesn't take advantage of the fact that the two lists are already sorted
  - It is going to have poor scalability and performance for very large lists

```
1     newlist = (xs + ys)
2     newlist.sort()
```

# MERGING SORTED LISTS

- Suppose we have two sorted lists ( $xs$  and  $ys$ )
- Devise an algorithm to merge them together into a single sorted list
- A **simple but inefficient** algorithm could be to simply **append the two lists together**, and **sort the result**
- But this doesn't take advantage of the fact that the **two lists are already sorted**
  - It is going to have poor scalability and performance for very large lists

```
1     newlist = (xs + ys)
2     newlist.sort()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/merge.py>

## BACK TO “A MORE REALISTIC PROBLEM”

- Let us go back now to our analysis of *Alice in Wonderland*
- Previously we sorted the words from the book, and eliminated duplicates
- Our vocabulary is also sorted
- So, find all items in the second list that are not in the first list, would be another way to implement `find_unknown_words`

⇒ <https://github.com/fpro-admin/lectures/blob/master/15/alice3.py>

# SUMMARY

- Let's review what we've done.

- We started with a word-by-word linear lookup in the vocabulary that ran in about 50 seconds
- We implemented a clever binary search, and got that down to 0.22 seconds, more than 200 times faster
- But then we did something even better: we sorted the words from the book, eliminated duplicates, and used a merging pattern to find words from the book that were not in the dictionary; this was about five times faster than even the binary lookup algorithm
- At the end, our algorithm is more than a 1000 times faster than our first attempt!

# EXERCISES

- Moodle activity at: [LE15: List Algorithms](#)

# PROGRAMMING FUNDAMENTALS

ANALYSIS OF ALGORITHMS

João Correia Lopes

INESC TEC, FEUP

22 November 2018

# CONTENTS

## 1 WHAT IS ALGORITHM ANALYSIS?

- B.1 Order of growth
- 2.3. Big-O Notation

## 2 PERFORMANCE OF PYTHON DATA STRUCTURES

- 2.6. Lists
- 2.7. Dictionaries
- 2.8. Summary

# ANALYSIS OF ALGORITHMS

- Analysis of algorithms is a branch of computer science that studies the performance of algorithms, especially their run time and space requirements ([Wikipedia](#))
- The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions
- Eric Schmidt jokingly asked Obama for “the most efficient way to sort a million 32-bit integers” and he quickly replied: “I think the bubble sort would be the wrong way to go” ([YouTube](#))

# PROBLEMS WHEN COMPARING ALGORITHMS

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

- The relative performance of the algorithms might depend on characteristics of the **hardware**
  - the general solution to this problem is to specify a machine model and analyze the number of steps, or operations, an algorithm requires under a given model
- Relative performance might depend on the **details of the dataset**
  - a common way to avoid this problem is to analyze the **worst case scenario**
- Relative performance also depends on the **size of the problem**
  - the usual solution to this problem is to express run time (or number of operations) as a function of problem size, and group functions into categories depending on how quickly they grow as problem size increases

# RUN TIME

- Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input:
  - Algorithm A takes  $T(n) = 100n + 1$  steps to solve a problem with size  $n$
  - Algorithm B takes  $T(n) = n^2 + n + 1$  steps to solve a problem with size  $n$
- The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

# ORDER OF GROWTH

- The **leading term** is the term with the highest exponent
- There will always be some value of  $n$  where  $an^2 > bn$ , for any values of  $a$  and  $b$
- For algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients
- An **order of growth** is a set of functions whose **growth behaviour** is considered equivalent
  - For example,  $2n$ ,  $100n$  and  $n + 1$  belong to the same order of growth
  - They are all linear

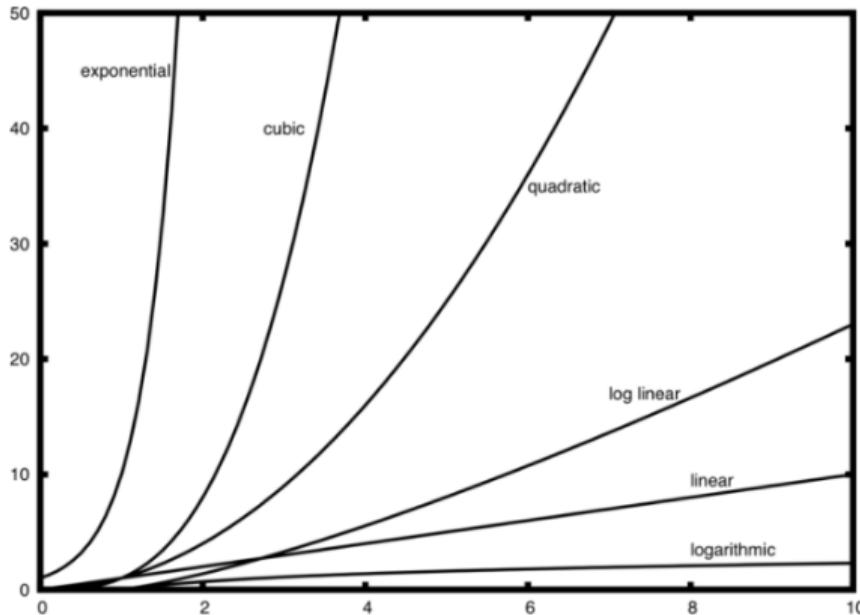
# BIG-O NOTATION

- $T(n)$  is the time it takes to solve a problem of size  $n$
- The **order of magnitude** function describes the part of  $T(n)$  that increases the fastest as the value of  $n$  increases
- Order of magnitude is often called **Big-O notation** (for “order”) and written as  $O(f(n))$
- It provides a **useful approximation to the actual number of steps in the computation**

# COMMON ORDER OF MAGNITUDE FUNCTIONS

<b>f(n)</b>	<b>Name</b>
1	Constant
$\log n$	Logarithmic
$n$	Linear
$n \log n$	Log Linear
$n^2$	Quadratic
$n^3$	Cubic
$2n$	Exponential

# COMMON ORDER OF MAGNITUDE FUNCTIONS



# COMPUTE $T(n)$

```
1     a=5
2     b=6
3     c=10
4     for i in range(n):
5         for j in range(n):
6             x = i * i
7             y = j * j
8             z = i * j
9             for k in range(n):
10                w = a*k + 45
11                v = b*b
12                d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

# COMPUTE $T(n)$

```
1     a=5
2     b=6
3     c=10
4     for i in range(n):
5         for j in range(n):
6             x = i * i
7             y = j * j
8             z = i * j
9         for k in range(n):
10            w = a*k + 45
11            v = b*b
12            d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

# COMPUTE $T(n)$

```
1     a=5
2     b=6
3     c=10
4     for i in range(n):
5         for j in range(n):
6             x = i * i
7             y = j * j
8             z = i * j
9         for k in range(n):
10            w = a*k + 45
11            v = b*b
12            d = 33
```

$$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$$

$$O(n^2)$$

# PERFORMANCE OF PYTHON DATA STRUCTURES

- Now that you have a general idea of Big-O notation and the differences between the different functions
- Let's talk about the Big-O performance for the operations on Python lists and dictionaries
- It is important for you to understand the efficiency of these Python data structures because they are the building blocks we will use as we implement other data structures
- The designers of Python had many choices to make when they implemented data structures

⇒ <https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython>

# LISTS

Operation	Big-O Efficiency
index []	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$

⇒ <https://github.com/fpro-admin/lectures/blob/master/16/timing.py>

# DICTIONARIES

As you probably recall, dictionaries differ from lists in that you can access items in a dictionary by a key rather than a position

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

# SUMMARY

- Algorithm analysis is an implementation-independent way of measuring an algorithm
- Big-O notation allows algorithms to be classified by their dominant process with respect to the size of the problem.

# EXERCISES

- Moodle activity at: [LE16: Analysis of Algorithms](#)

# PROGRAMMING FUNDAMENTALS

## RECUSION

João Correia Lopes

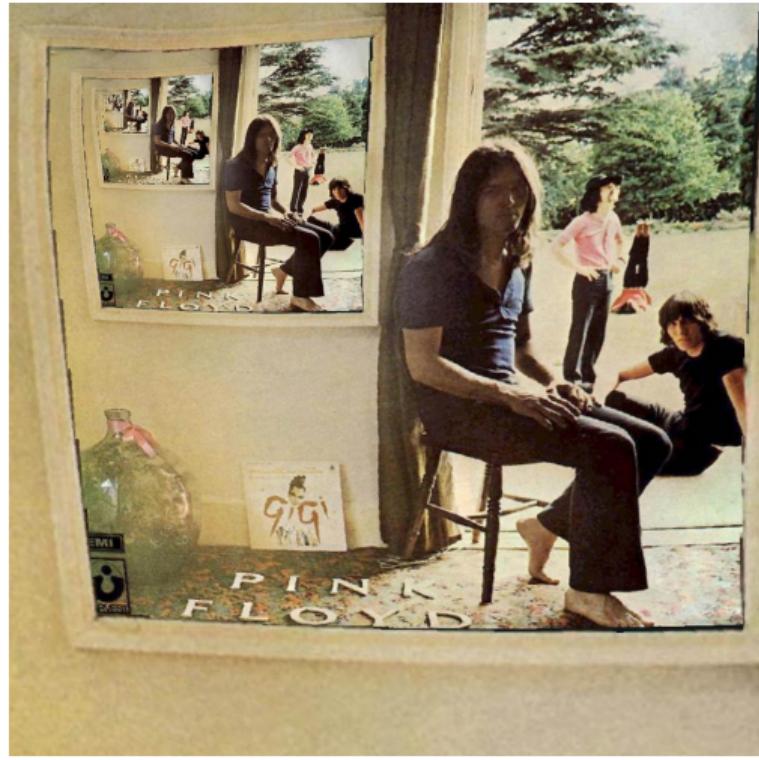
INESC TEC, FEUP

27 November 2018

# CONTENTS

## 1 RECURSION

- Case study: Factorial
- Scope of a recursive function
- 10.1 Drawing Fractals
- 10.2 Recursive data structures
- 10.3 Processing recursive number lists
- Infinite Recursion
- 10.4 Case study: Fibonacci numbers
- 10.5 Example with recursive directories and files
- 10.7 Mutual Recursion



# RECURSION

- **Recursion** means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective
- Recursion is a method of solving problems that involves **breaking a problem down into smaller and smaller sub-problems** until you get to a small enough problem that it can be solved trivially
- Programming languages generally support recursion, which means that, in order to solve a problem, **functions can call themselves** to solve smaller sub-problems
- Recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program <sup>1</sup>

---

<sup>1</sup>Any problem that can be solved iteratively (with a for or while loop) can also be solved recursively. However, recursion takes a while wrap your head around, and because of this, it is generally only used in specific cases, where either your problem is recursive in nature, or your data is recursive

# FACTORIAL

```
1 n! = n * (n-1) * (n-2) * (n-3) * ... * 1
```

- **Base case:** we know the factorial of 1<sup>2</sup>

```
1 if n == 1:  
2     return 1
```

- **Recursive step:** Rewrite in terms of something simpler to reach base case

```
1 else:  
2     return n * factorial(n-1)
```

---

<sup>2</sup>Without a base case, you'll have **infinite recursion**, and your program will not work.

# FACTORIAL

```
1 n! = n * (n-1) * (n-2) * (n-3) * ... * 1
```

- **Base case:** we know the factorial of  $1^2$

```
1 if n == 1:  
2     return 1
```

- **Recursive step:** Rewrite in terms of something simpler to reach base case

```
1 else:  
2     return n * factorial(n-1)
```

<sup>2</sup>Without a base case, you'll have **infinite recursion**, and your program will not work.

# FACTORIAL

```
1 n! = n * (n-1) * (n-2) * (n-3) * ... * 1
```

- **Base case:** we know the factorial of 1<sup>2</sup>

```
1 if n == 1:  
2     return 1
```

- **Recursive step:** Rewrite in terms of something simpler to reach base case

```
1 else:  
2     return n * factorial(n-1)
```

<sup>2</sup>Without a base case, you'll have **infinite recursion**, and your program will not work.

# FACT (3)

## Recursion 1

Here, n=3

```
def fact( 3 ):  
    if 3 == 1:  
        return 1  
    else :  
        return (3 * fact( 2 ))
```

Return to main function

## Recursion 2

```
def fact ( 2 ):  
    if 2 == 1:  
        return 1  
    else :  
        return ( 2 * fact( 1 ))
```

## Recursion 3

```
def fact ( 1 ):  
    if 1 == 1:  
        return 1  
    else :  
        return (n * fact( n-1 ))
```

# SCOPE OF A RECURSIVE FUNCTION

- See the scope in [Python Tutor](#)

```
1  def fact(n):
2      if n == 1:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  print(fact(4))
```

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

# SCOPE OF A RECURSIVE FUNCTION

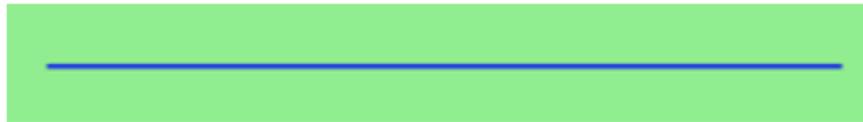
- See the scope in [Python Tutor](#)

```
1  def fact(n):
2      if n == 1:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  print(fact(4))
```

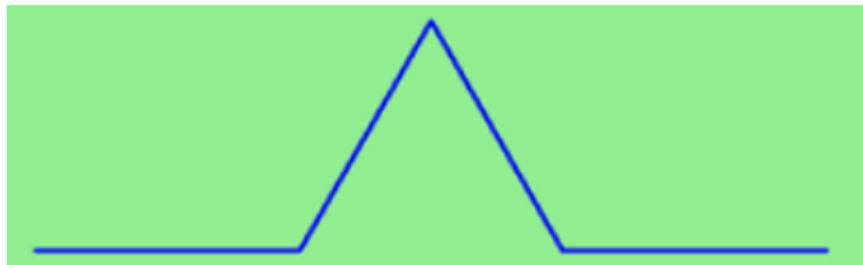
- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

# KOCH FRACTAL

- A **fractal** is a drawing which also has self-similar structure, where it can be defined in terms of itself
- This is a typical example of a problem which is recursive in nature

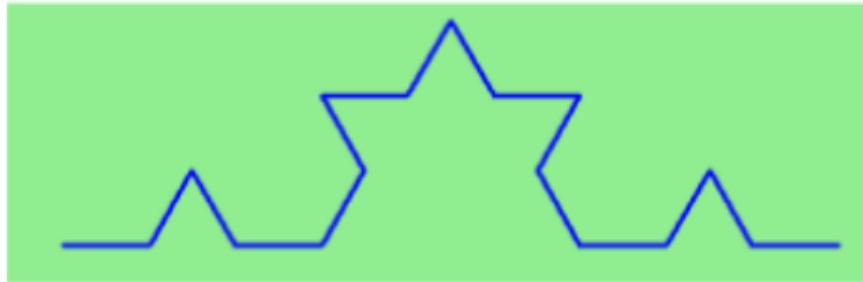


An order 0 Koch fractal

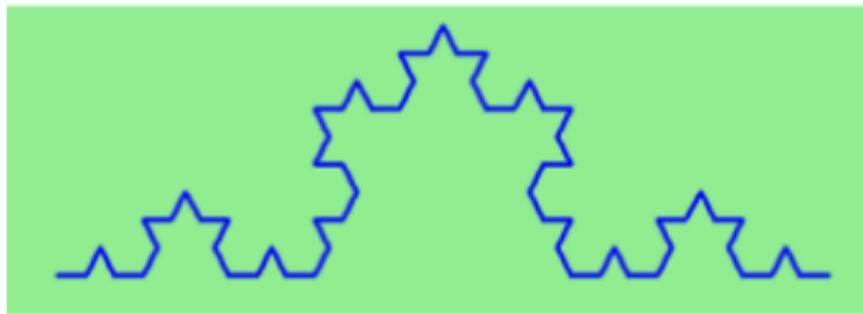


An order 1 Koch fractal

# KOCH FRACTAL



An order 2 Koch fractal



An order 3 Koch fractal

# RECURSION, THE HIGH-LEVEL VIEW

- The function works correctly when you call it for an order 0 fractal
- Focus on is how to draw an order 1 fractal *if I can assume the order 0 one is already working*
- You're practicing **mental abstraction** — ignoring the subproblem while you solve the **big problem**
- See that it will work when called for order 2 *under the assumption that it is already working for level 1*
- And, in general, if I can assume the order  $n-1$  case works, can I just solve the level  $n$  problem?
- Students of mathematics who have played with proofs of **induction** should see some very strong similarities here

# RECURSION, THE LOW-LEVEL OPERATIONAL VIEW

- The trick of “unrolling” the recursion gives us an operational view of what happens
- You can trace the program into `koch_3`, and from there, into `koch_2`, and then into `koch_1`, etc., all the way down the different layers of the recursion.

⇒ [https://github.com/fpro-admin/lectures/blob/master/17/low\\_level.py](https://github.com/fpro-admin/lectures/blob/master/17/low_level.py)

# RECURSIVE DATA STRUCTURES

- The organization of data for the purpose of making it easier to use is called a **data structure**
- Most of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways
- Lists and tuples can also be nested, providing many possibilities for organizing data
- Example: A *nested number list* is a list whose elements are either:
  - 1 numbers
  - 2 nested number lists
- Notice that the term, *nested number list* is used in its own definition
- Recursive definitions like this provide a concise and powerful way to describe **recursive data structures**

# PROCESSING RECURSIVE NUMBER LISTS

```
1 >>> sum([1, 2, 8])
2 11
3
4 >>> sum([1, 2, [11, 13], 8])
5 Traceback (most recent call last):
6   File "<interactive input>", line 1, in <module>
7 TypeError: unsupported operand type(s) for +: 'int' and 'list'
8 >>>
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/17/rec\\_sum.py](https://github.com/fpro-admin/lectures/blob/master/17/rec_sum.py)

# INFINITE RECURSION

```
1 def recursion_depth(number):
2     print("{0}, ".format(number), end="")
3     recursion_depth(number + 1)
4
5 recursion_depth(0)
6
7 ...
8
9 RuntimeError: maximum recursion depth exceeded ...
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/17/infinite\\_recursion.py](https://github.com/fpro-admin/lectures/blob/master/17/infinite_recursion.py)

# CASE STUDY: FIBONACCI NUMBERS

- Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, ...
- was devised by Fibonacci (1170-1250) who used this to model the breeding of (pairs) of rabbits
  - If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have  $13+21=34$ , of which 21 are adults.

```
1 fib(0) = 0
2 fib(1) = 1
3 fib(n) = fib(n-1) + fib(n-2)
4 for n >= 2
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/17/fib.py>

# RECURSIVE DIRECTORIES AND FILES

- Let's do a program that lists the contents of a directory and all its sub-directories
- First we need `get_dirlist(path)`

```
1 import os
2
3 def get_dirlist(path):
4     """
5         Return a sorted list of all entries in path.
6         This returns just the names, not the full path to the names.
7     """
8     dirlist = os.listdir(path)
9     dirlist.sort()
10
11     return dirlist
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/17/dirs.py>

# MUTUAL RECURSION

- In addition to a function calling just itself, it is also possible to make multiple functions that call each other
- This is rarely really useful, but it can be used to make state machines
- In mathematics, a Hofstadter sequence is a member of a family of related integer sequences defined by non-linear recurrence relations
- The *Hofstadter Female and Male sequences*:

```
1 F ( 0 ) = 1
2 M ( 0 ) = 0
3 F ( n ) = n - M ( F ( n - 1 ) ), n > 0
4 M ( n ) = n - F ( M ( n - 1 ) ), n > 0
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/17/hofstadter.py>

# EXERCISES

- Moodle activity at: [LE17: Recursion](#)

# PROGRAMMING FUNDAMENTALS

## MORE RECURSION

João Correia Lopes

INESC TEC, FEUP

29 November 2018

# CONTENTS

## 1 CASE STUDY: TOWER OF HANOI

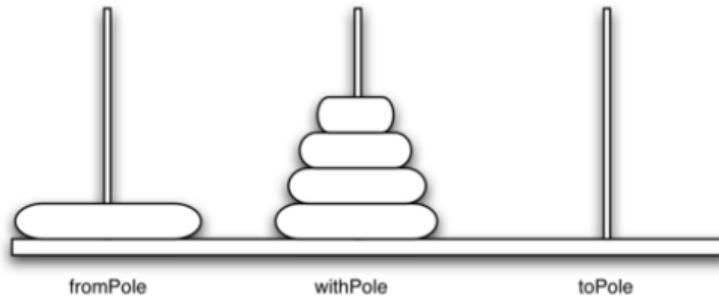
## 2 ITERATION VERSUS RECURSION

- Calculating the Sum of a List of Numbers
- Factorial
- Fibonacci
- Is a Palindrome
- Converting to any Base

## 3 SUMMARY

# TOWER OF HANOI

- The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883  
(⇒ [Wikipedia](#))
- The priests were given three poles and a stack of 64 gold disks
- Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints:
  - They could only move one disk at a time, and
  - They could never place a larger disk on top of a smaller one



<http://interactivepython.org/runestone/static/pythonds/Recursion/TowerofHanoi.html>

## TOWER OF HANOI (2)

- The number of moves required to correctly move a tower of 64 disks is

$$2^{64} - 1 = 18446744073709551615$$

- At a rate of one move per second, it takes: 584 942 417 355 years!

■ → [Tower of Hanoi | GeeksforGeeks](#)

■ Pseudo-code:

- *Move a tower of height – 1 from the original pole to the intermediate pole, using the final pole*
- *Move the remaining disk to the final pole*
- *Move the tower of height – 1 from the intermediate pole to the final pole using the original pole*

⇒ <https://github.com/fpro-admin/lectures/blob/master/18/hanoi.py>

## TOWER OF HANOI (2)

- The number of moves required to correctly move a tower of 64 disks is

$$2^{64} - 1 = 18446744073709551615$$

- At a rate of one move per second, it takes: 584 942 417 355 years!
- ⇒ [Tower of Hanoi | GeeksforGeeks](#)

- Pseudo-code:

- Move a tower of height – 1 from the original pole to the intermediate pole, using the final pole
  - Move the remaining disk to the final pole
  - Move the tower of height – 1 from the intermediate pole to the final pole using the original pole

⇒ <https://github.com/fpro-admin/lectures/blob/master/18/hanoi.py>

## TOWER OF HANOI (2)

- The number of moves required to correctly move a tower of 64 disks is

$$2^{64} - 1 = 18446744073709551615$$

- At a rate of one move per second, it takes: 584 942 417 355 years!

- ⇒ [Tower of Hanoi | GeeksforGeeks](#)

- Pseudo-code:

- 1 *Move a tower of height – 1 from the original pole to the intermediate pole, using the final pole*
- 2 *Move the remaining disk to the final pole*
- 3 *Move the tower of height – 1 from the intermediate pole to the final pole using the original pole*

⇒ <https://github.com/fpro-admin/lectures/blob/master/18/hanoi.py>

# ITERATION VS. RECURSION

- Recursion and iteration perform the same kinds of tasks:
  - Solve a complicated task one piece at a time, and combine the results
- Emphasis of iteration:
  - keep repeating until a task is finished
  - e.g. loop counter reaches limit, list reaches the end, ...
- Emphasis of recursion:
  - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it;  
combine the results
  - e.g. recursive factorial function

# SUM OF A LIST OF NUMBERS

- We will begin our investigation with a simple problem that you already know how to solve without using recursion
- Suppose that you want to calculate the sum of a list of numbers such as:

[1, 3, 5, 7, 9]

# SUM OF A LIST OF NUMBERS **ITERATIVE**

- The function uses an **accumulator** variable (`the_sum`) to compute a running total of all the numbers in the list by **starting with 0** and **adding each number in the list**

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/listsum\\_iter.py](https://github.com/fpro-admin/lectures/blob/master/18/listsum_iter.py)

# SUM OF A LIST OF NUMBERS **RECURSIVE**

- The sum of a list of length 1 is **trivial**; it is just the number in the list
- The series of (recursive) calls may be seen as a **series of simplifications**

$$(1 + (3 + (5 + (7 + 9)))))$$

- Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/listsum\\_rec.py](https://github.com/fpro-admin/lectures/blob/master/18/listsum_rec.py)

# FACTORIAL RECURSIVE

```
1  def fact_rec(n):
2      """ assume n >= 0 """
3      if n <= 1:
4          return 1
5      else:
6          return n * fact_rec(n-1)
```

- O(n)
- Look at tail recursion ( $\Rightarrow$  Neopythonic)

# FACTORIAL **RECURSIVE**

```
1  def fact_rec(n):
2      """ assume n >= 0 """
3      if n <= 1:
4          return 1
5      else:
6          return n * fact_rec(n-1)
```

- O(n)
- Look at **tail recursion** ( $\Rightarrow$  [Neopythonic](#))

# FACTORIAL ITERATIVE

```
1 def fact_iter(n):
2     prod = 1
3     for i in range(1, n+1):
4         prod = i * prod
5     return prod
```

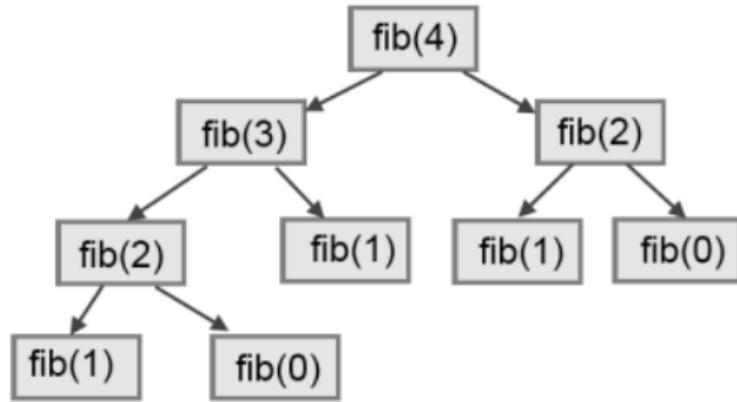
- $O(n)$
- Is it easier to read?
- Is it faster?

# FACTORIAL ITERATIVE

```
1 def fact_iter(n):
2     prod = 1
3     for i in range(1, n+1):
4         prod = i * prod
5     return prod
```

- O(n)
- Is it easier to read?
- Is it faster?

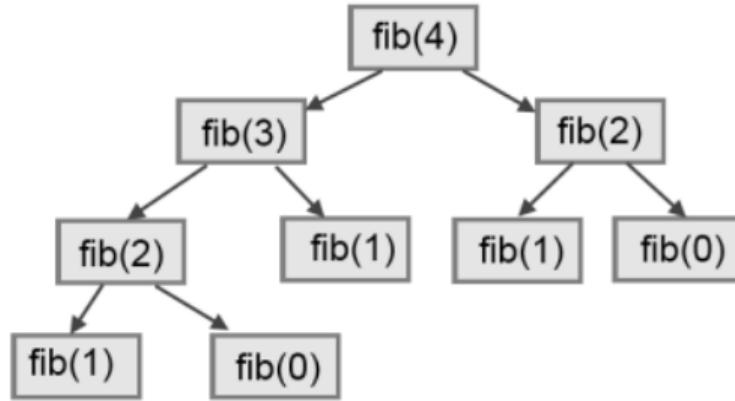
# FIBONACCI RECURSIVE



- $O(2^n)$
- It is a **binary tree** of height  $n$ : for  $n = 4$  we have  $2^n - 1 = 15$  nodes
- ⇒ [StackExchange](#)

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/fib\\_rec.py](https://github.com/fpro-admin/lectures/blob/master/18/fib_rec.py)

# FIBONACCI **RECURSIVE**



- $O(2^n)$
- It is a **binary tree** of height  $n$ : for  $n = 4$  we have  $2^n - 1 = 15$  nodes
- ⇒ [StackExchange](#)

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/fib\\_rec.py](https://github.com/fpro-admin/lectures/blob/master/18/fib_rec.py)

# FIBONACCI EFFICIENT

- Calling `fib(34)` results in **11 405 773** recursive calls
- Calling `fib_efficient(34)` results in **65** recursive calls
- Using dictionaries to capture intermediate results can be very efficient (**memoisation**)

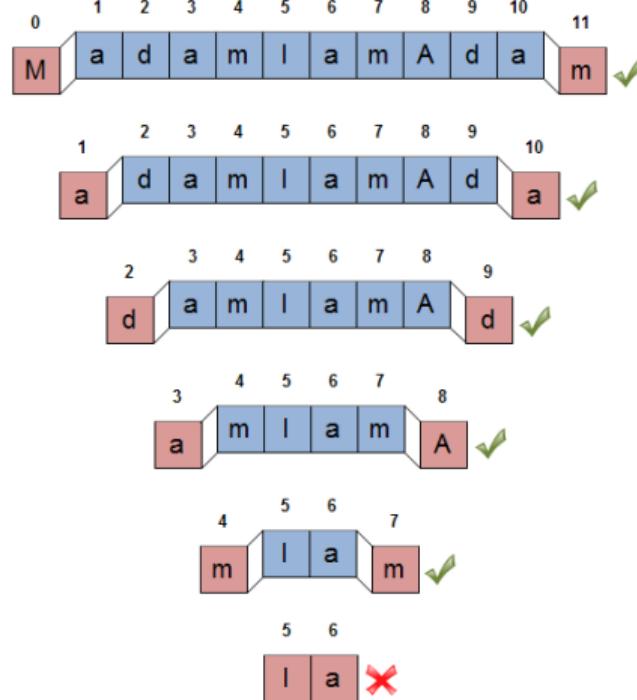
⇒ [https://github.com/fpro-admin/lectures/blob/master/18/fib\\_efficient.py](https://github.com/fpro-admin/lectures/blob/master/18/fib_efficient.py)

# FIBONACCI **ITERATIVE**

- $O(n)$  — it's a for cycle

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/fib\\_iter.py](https://github.com/fpro-admin/lectures/blob/master/18/fib_iter.py)

# IS A PALINDROME **RECURSIVE**



⇒ [https://github.com/fpro-admin/lectures/blob/master/18/is\\_palindrome\\_rec.py](https://github.com/fpro-admin/lectures/blob/master/18/is_palindrome_rec.py)

# IS A PALINDROME

## ITERATIVE

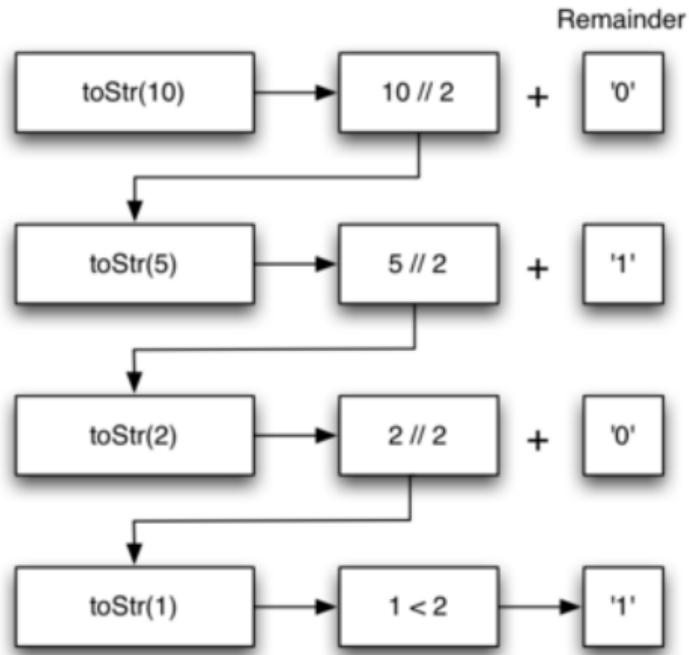
- $O(n)$  — complexity of the `join` method

⇒ [https://github.com/fpro-admin/lectures/blob/master/18/is\\_palindrome\\_iter.py](https://github.com/fpro-admin/lectures/blob/master/18/is_palindrome_iter.py)

# CONVERTING AN INTEGER TO A STRING IN ANY BASE

- Suppose you want to convert an integer to a string in some base between binary and hexadecimal
- While there are many approaches one can take to solve this problem, the recursive formulation of the problem is very elegant:
  - 1 *Reduce the original number to a series of single-digit numbers*
  - 2 *Convert the single digit-number to a string using a lookup*
  - 3 *Concatenate the single-digit strings together to form the final result*

# CONVERTING AN INTEGER TO BASE 10



⇒ [https://github.com/fpro-admin/lectures/blob/master/18/to\\_base.py](https://github.com/fpro-admin/lectures/blob/master/18/to_base.py)

# RECURSION VS. ITERATION

## ■ Advantages of Python Recursion

- Recursive functions make the code look clean and elegant
- Very flexible in data structure like *tree traversals, stacks, queues, linked list*
- Big and complex iterative solutions are easy and simple with Python recursion
- Sequence generation is easier with recursion than using some nested iteration
- Algorithms can be defined recursively making it much easier to visualize and prove

## ■ Disadvantages of Python Recursion

- Sometimes the logic behind recursion is hard to follow
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time<sup>1</sup>
- More difficult to trace and debug
- Recursive functions often throw a *Stack Overflow Exception* when processing or operations are too large

---

<sup>1</sup>For every recursive call separate memory is allocated for the variables (the Activation Record aka Stack Frame).

# SUMMARY ABOUT RECURSION

- All recursive algorithms must have a base case
- A recursive algorithm must change its state and make progress toward the base case
- A recursive algorithm must call itself (recursively)
- Recursion can take the place of iteration in some cases
- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve
- Recursion is not always the answer: sometimes a recursive solution may be more computationally expensive than an alternative algorithm.

# EXERCISES

- Moodle activity at: [LE18: More recursion](#)

# PROGRAMMING FUNDAMENTALS

## FUNCTIONAL PROGRAMMING WITH COLLECTIONS

João Correia Lopes

INESC TEC, FEUP

4 december 2018

# CONTENTS

## 1 FUNCTIONAL PROGRAMMING W/ COLLECTIONS

- Lists of Tuples
- List Comprehensions
- Sequence Processing Functions: `map()`, `filter()`
- Advanced List Sorting
- The Lambda

# FUNCTIONAL PROGRAMMING

- “Programming in a functional language consists in building definitions and using the computer to evaluate expressions.”<sup>1</sup>
- The primary role of the programmer is to construct a function to solve a give problem.
- This function, which may involve a number of subsidiary functions, is expressed in notation that obeys normal mathematical principles.
- The primary role of the computer is to act as an evaluator or calculator: its job is to evaluate expressions and print results.

---

<sup>1</sup>Bird & Wadler, Introduction to Functional Programming, Prentice-Hall, 1988

# ADVANCED COLLECTION CONCEPTS

- *Lists of Tuples* — describe the relatively common Python data structure built from a list of tuples
- *List Comprehensions* — powerful list construction method used to simplify some common list-processing patterns
- `map()`, `filter()` — functions that can simplify some list processing and provide features that overlap with list comprehensions
- *lambda forms* — aren't essential for Python programming, but they're handy for clarifying a piece of code in some cases

# LISTS OF TUPLES

- The list of tuple structure is remarkably useful
- One common situation is processing list of simple coordinate pairs for 2-dimensional or 3-dimensional geometries
- As an example of using a red, green, blue **tuple**, we may have a list of individual colors that looks like:

```
1 colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x10,0xff,0xff) ]
```

# WORKING WITH LISTS OF TUPLES

- In dictionaries, the `dict.items()` method provides the dictionary keys and values as a list of 2-tuples
- The `zip()` built-in function *interleaves* two or more sequences to create a list of tuples
- A interesting form of the `for` statement is one that exploits multiple assignment to work with a list of tuples

```
1  for c,f in [("red",18), ("black",18), ("green",2)]:  
2      print("{0} occurs {1}".format(c, f/38.0))
```

- The `for` statement uses a form of multiple assignment to split up each tuple into a fixed number of variables

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/for.py>

# LIST DISPLAYS

- For constructing a list, a set or a dictionary, Python provides special syntax called “displays”<sup>2</sup>
- The most common list *display* is the simple literal value:

```
1 [ expression <, ... > ]
```

- For example:

```
1 fruit = ["Apples", "Peaches", "Pears", "Bananas"]
```

- But Python has a second kind of list *display*, based on a list comprehension

---

<sup>2</sup>The Python Language Reference

# LIST COMPREHENSIONS

- A list comprehension is an expression that combines a function, a for statement, and an optional if statement
- This allows a simple, clear expression of the processing that will build up an iterable sequence
- The most important thing about a list comprehension is that it is an iterable that applies a calculation to another iterable
- A list display can use a list comprehension iterable to create a new list

1

```
even = [2*x for x in range(18)]
```

# LIST COMPREHENSION SEMANTICS

- When we write a list comprehension, we will provide an iterable, a variable and an expression
- Python will process the iterator as if it was a for-loop, iterating through a sequence of values
- It evaluates the expression, once for each iteration of the for-loop
- The resulting values can be collected into a fresh, new list, or used anywhere an iterator is used

```
1     string = "Hello 12345 World"
2     for n in [int(x) for x in string if x.isdigit()]:
3         print(n*n)
```

# LIST COMPREHENSION SYNTAX

- A list comprehension is — technically — a complex expression
- It's often used in list displays, but can be used in a variety of places where an iterator is expected

```
1     expr <for-clause>
```

- The `expr` is any expression
- It can be a simple constant, or any other expression (including a nested list comprehension)
- The `for-clause` mirrors the `for` statement

```
1     for variable in sequence
```

# COMPREHENSION IN A LIST DISPLAY

- For example:

```
1 even = [2*x for x in range(18)]
2 hardways = [(x,x) for x in (2,3,4,5)]
3 samples = [random.random() for x in range(10)]
```

- A list display that uses a list comprehension behaves like the following loop:

```
1 r = []
2 for variable in sequence:
3     r.append(expr)
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/19/for\\_comp.py](https://github.com/fpro-admin/lectures/blob/master/19/for_comp.py)

# COMPREHENSIONS OUTSIDE LIST DISPLAYS

- We can use the iterable list comprehension in other contexts that expect an iterator

```
1 square = sum((2*a+1) for a in range(10))
2
3 column_1 = tuple(3*b+1 for b in range(12))
4
5 # create a generator object that will iterate over 100 values
6 rolls = ((random.randint(1,6), random.randint(1,6)) for u in range(100))
7
8 hardways = any(d1==d2 for d1,d2 in rolls)
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/19/out\\_comp.py](https://github.com/fpro-admin/lectures/blob/master/19/out_comp.py)

# THE IF CLAUSE

- A list comprehension can also have an **if-clause**

```
1     expr <for-clause> <if-clause>
```

- Here is an example of a complex list comprehension in a list display

```
1     hardways = [ (x,x) for x in range(1,7) if x+x not in (2, 12) ]
```

- This more complex list comprehension behaves like the following loop:

```
1     r= []
2     for variable in sequence :
3         if filter:
4             r.append( expr )
```

## ANOTHER EXAMPLE

```
1      >>> [ (x, 2*x+1) for x in range(10) if x % 3 == 0]
2
3      [(0, 1), (3, 7), (6, 13), (9, 19)]
```

### ■ This works as follows:

- 1 The for-clause iterates through the 10 values given by `range(10)`, assigning each value to the local variable `x`
- 2 The if-clause evaluates the filter function, `x % 3 == 0`. If it is `False`, the value is skipped; if it is `True`, the expression, at `(x, 2*x+1)`, is evaluated and retained
- 3 The sequence of 2-tuples are assembled into a list

# NESTED LIST COMPREHENSIONS

- A list comprehension can have any number of *for-clauses* and *if-clauses*, freely-intermixed
- A *for-clause* must be first
- The clauses are evaluated from left to right
- ⇒ [The Python Language Reference](#)

```
1  # given a matrix 3x4 (see code)
2  [[row[i] for row in matrix] for i in range(4)]
3
4  transposed = []
5  for i in range(4):
6      transposed.append([row[i] for row in matrix])
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/transpose.py>

## MAP(), FILTER()

- The `map()` and `filter()` built-in functions are handy functions for processing sequences without writing lengthy `for`-loops
- The idea of each is **to take a small function you write and apply it to all the elements of a sequence**, saving you from writing an explicit loop
- The implicit loop within each of these functions may be faster than an explicit `for` loop
- Additionally, each of these is a *pure function*, returning a *result value*
- This allows the results of the functions to be combined into complex expressions relatively easily

# PROCESSING PIPELINE

- It is very, very common to apply a single function to every value of a list
- In some cases, we may apply multiple simple functions in a kind of “processing pipeline”

```
1  # NBA's players heights in (feet, inch)
2  heights = [(5,8), (5,9), (6,2), (6,1), (6,7)]
3
4  # convert (feet, inch) to inches
5  def ftin_2_in(ftin):
6      feet, inches = ftin
7      return 12.0*feet + inches
8
9  map(ftin_2_in, heights)
10 ...
11
12 # now convert inches to meters
13 map(in_2_m, map(ftin_2_in, heights))
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/19/metric\\_sizes.py](https://github.com/fpro-admin/lectures/blob/master/19/metric_sizes.py)

# MAP

- Create a new iterator from the results of applying the given *function* to the items of the the given *sequence*

```
1 map(function, sequence, [...])
```

- This function behaves as if it had the following definition:

```
1 def map(a_function, a_sequence):  
2     return [a_function(v) for v in a_sequence]
```

```
1 >>> list(map(int, ["10", "12", "14", 3.1415926, 5]))  
2 [10, 12, 14, 3, 5]
```

- If more than one sequence is given, the corresponding items from each sequence are provided as arguments to the function (None used for missing values, as in zip())

# FILTER

- Return a iterator containing those items of sequence for which the given function is True
- If the function is None, return a list of items that are equivalent to True

```
1 filter(function, sequence)
```

- This function behaves as if it had the following definition:

```
1 def filter(a_function, a_sequence):  
2     return [v for v in a_sequence if a_function(v)]
```

```
1 >>> def over_2(m):  
2     ...     return m > 2.0  
3 >>> list(filter(over_2, map(in_2_m, map(ftin_2_in, heights))))  
4 [2.01]
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/filter.py>

# FILTER

- Return a iterator containing those items of sequence for which the given function is True
- If the function is None, return a list of items that are equivalent to True

```
1 filter(function, sequence)
```

- This function behaves as if it had the following definition:

```
1 def filter(a_function, a_sequence):  
2     return [v for v in a_sequence if a_function(v)]
```

```
1 >>> def over_2(m):  
2 ...     return m > 2.0  
3 >>> list(filter(over_2, map(in_2_m, map(ftin_2_in, heights))))  
4 [2.01]
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/filter.py>

# REDUCE

## ■ Removed in Python3!

- Use `functools.reduce()` if you really need it
- However, 99 percent of the time an explicit `for` loop is more readable
- The idea is to apply the given function to an internal accumulator and each item of a sequence, from left to right, so as to reduce the sequence to a single value

```
1  def reduce(a_function, a_sequence, init= 0):  
2      r = init  
3      for s in a_sequence:  
4          r = a_function(r, s)  
5      return r
```

- built-in `sum()`, `any()` and `all()` are kinds of reduce functions

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/reduce.py>

# ZIP

- The `zip()` function interleaves values from two or more sequences to create a new sequence
- The new sequence is a sequence of tuples
- Each item of a tuple is the corresponding values from each sequence
- If any sequence is too long, truncate it

```
1     zip(sequence, [sequence...])
```

- Here's an example:

```
1     list(zip(range(5), range(1,12,2) ))
2     [(0, 1), (1, 3), (2, 5), (3, 7), (4, 9)]
```

# LIST SORTING

- Consider a list of tuples (that came from a spreadsheet csv file )

```
1 job_data = [
2     ('121','Wyoming','NY',8722),
3     ('123','Yates','NY',5094)
4     ...
5     ('001','Albany','NY',162692),
6     ('003','Allegany','NY',11986),
7 ]
```

- Sorting this list can be done trivially with the `list.sort()` method

```
1 job_data.sort()
```

- This kind of sort will simply compare the tuple items in the order presented in the tuple
- In this case, the country number is first

# SORTING WITH KEY EXTRACTION

- What if we want to sort by some other column, like state name or jobs?
- The `sort()` method of a list can accept a keyword parameter, `key`, that provides a key extraction function
- This function returns a value which can be used for comparison purposes
- To sort our `job_data` by the third field, we can use a function like the following:

```
1  def by_state(a):
2      return a[1]
3
4  job_data.sort(key=by_state)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/sort.py>

# THE LAMBDA

- The functions `map()`, `filter()` and the `list.sort()` method all use small functions to control their operations
- Instead of defining a function, Python allows us to provide a *lambda form*
- This is a kind of **anonymous, one-use-only function body** in places where we only need a very, very simple function
- A ***lambda* form** is like a defined function: it has parameters and computes a value
- The body of a *lambda*, however, **can only be a single expression**, limiting it to relatively simple operations

```
1 lambda a: a[0]*2+a[1]    # define a lambda
2
3 (lambda n: n*n)(5)       # define a lambda and call it
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/lambda.py>

# PARAMETERIZING A LAMBDA

- Sometimes we want to have a *lambda* with an argument defined by the “context” or “scope” in which the *lambda* is being used

```
1  >>> def timesX(x):
2      return lambda a: x*a
3
4  >>> t2 = timesX(2)      # a function that multiplies the arg. by 2
5  >>> t2(5)
6  10
7
8  >>> t3 = timesX(3)      # a function that multiplies the arg. by 3
9  >>> t3(5)
10 15
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/19/lambda.py>

# EXERCISES

- Moodle activity at: [LE18: FP with Collections](#)

# PROGRAMMING FUNDAMENTALS

EFFECT-FREE PROGRAMMING STYLE

João Correia Lopes

INESC TEC, FEUP

6 december 2018

# CONTENTS

## 1 EFFECT-FREE PROGRAMMING STYLE

- Python & Functional Programming
- 4.17.1 Modifiers vs Pure Functions
- Iterators
- (Avoiding) Flow Control
- Closures and Callable Instances
- Generators and Lazy Evaluation
- Utility Higher-Order Functions

# EFFECT-FREE PROGRAMMING STYLE

- Function calls have **no side effects** and variables are **immutable**
  - Do not use `global` and `nonlocal` statements
  - Take care about data types that are mutable
  - Do not use Input/Output

# PYTHON & FUNCTIONAL PROGRAMMING

*Python is most definitely not a “pure functional programming language”; side effects are widespread in most Python programs. That is, variables are frequently rebound, mutable data collections often change contents, and I/O is freely interleaved with computation.*

*It is also not even a “functional programming language” more generally.*

*However, Python is a multiparadigm language that makes functional programming easy to do when desired, and easy to mix with other programming styles.<sup>1</sup>*

---

<sup>1</sup>David Mertz, Functional Programming in Python, O'Reilly Media, 2015

# FUNCTIONAL PROGRAMMING

- In a functional program, input flows through a set of functions
- Each function operates on its input and produces some output
- Functional style discourages functions with *side effects* that *modify internal state* or make other changes that aren't visible in the function's return value
- Functions that have no side effects at all are called **purely functional**
- Avoiding side effects means not using data structures that get updated as a program runs; every function's output must only depend on its input

# ADVANTAGES OF THE FUNCTIONAL STYLE

Why should you avoid objects (OOP) and side effects?

- There are theoretical (T) and practical (P) advantages to the functional style:
  - Formal provability (T)
  - Modularity (P)
  - Ease of debugging and testing (P)
  - Composability (P)

## EFFECT-FREE CODE

The advantage of a pure function and side-effect free code is that it is generally easier to debug and test.

# ADVANTAGES OF THE FUNCTIONAL STYLE

Why should you avoid objects (OOP) and side effects?

- There are theoretical (T) and practical (P) advantages to the functional style:
  - Formal provability (T)
  - Modularity (P)
  - Ease of debugging and testing (P)
  - Composability (P)

## EFFECT-FREE CODE

The advantage of a pure function and side-effect free code is that it is generally easier to debug and test.

# MODIFIERS VS PURE FUNCTIONS (RECAP)

- Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**
- A **pure function** does not produce side effects
  - It communicates with the calling program only through parameters, which it does not modify, and a return value
- Is `double_stuff()` pure?

```
1  def double_stuff(values):  
2      """ Double the elements of values """  
3      for index, value in enumerate(values):  
4          values[index] = 2 * value
```

# ITERATORS (RECAP)

- Iterators are an important foundation for writing functional-style programs
- An iterator is an object representing a stream of data and returns the data one element at a time
- Several of Python's built-in data types support iteration, the most common being lists and dictionaries
- An object is called **iterable** if you can **get an iterator for it**
- Python expects iterable objects in several different contexts, the most important being the `for` statement
- Iterators can be **materialised** as lists or tuples by using the `list()` or `tuple()` constructor functions
- Built-in functions such as `max()` and `min()` can take a single iterator argument
- The `in` and `not in` operators also support iterators
- Note that you can **only go forward in an iterator**; there's no way to get the previous element, reset the iterator, or make a copy of it

# IMPERATIVE PYTHON PROGRAMS

- “In typical imperative Python programs<sup>2</sup> a block of code generally consists of some outside loops (`for` or `while`), assignment of state variables within those loops, modification of data structures like `dicts`, `lists`, and `sets` (or various other structures, either from the standard library or from third-party packages), and some branch statements (`if/elif/else` or `try/except/finally`).”
- The imperative flow control described in the last paragraph is much more about the “how” than the “what” and **we can often shift the question**

---

<sup>2</sup>Including those that make use of classes and methods to hold their imperative code.

# COMPREHENSIONS

- Using comprehensions is often a way both to make code more compact and to *shift our focus from the “how” to the “what”*
- A comprehension is an expression that uses the same keywords as loop and conditional blocks, but inverts their order to **focus on the data rather than on the procedure**
- Simply changing the form of expression can often make a surprisingly large difference in how we *reason about code* and how easy it is to understand it
- Python includes: *List comprehensions, Generator comprehensions, Set comprehensions, and Dictionary comprehensions*

# MENTAL SHIFT

- The *ternary operator* also performs a similar restructuring of our focus, using the same keywords in a different order
- For example, if our original code was:

```
1 collection = list()
2 for datum in data_set:
3     if condition(datum):
4         collection.append(datum)
5     else:
6         new = modify(datum)
7         collection.append(new)
```

- Somewhat more compactly we could write this as:

```
1 collection = [d if condition(d) else modify(d)
2                      for d in data_set]
```

# GENERATORS

- **Generator comprehensions** have the same syntax as list comprehensions — other than that there are no square brackets around them (but parentheses are needed syntactically in some contexts, in place of brackets)
- They are also *lazy*
- That is to say that they are merely a description of “how to get the data” that is not realised until one explicitly asks for it, either by calling `.next()` on the object, or by looping over it

```
1 log_lines = (line for line in read_line(huge_log_file)
2                               if complex_condition(line))
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/20/generators.py>

# RECURSION (RECAP)

- Functional programmers often put weight in **expressing flow control** through recursion rather than through loops
- Done this way, we can **avoid altering the state** of any variables or data structures within an algorithm, and more importantly get more at the “what” than the “how” of a computation
- In the cases where recursion is just “iteration by another name”, iteration is more “Pythonic”
- Where recursion is compelling, and sometimes even the only really obvious way to express a solution, is when a problem offers itself to a “divide and conquer” approach (i.e., a problem can readily be partitioned into smaller problems)

⇒ <https://github.com/fpro-admin/lectures/blob/master/20/factorialR.py>

# QUICKSORT

- For example, the *quicksort algorithm* is very elegantly expressed without any state variables or loops, but wholly through recursion

```
1  def quicksort(lst):
2      "Quicksort over a list-like sequence"
3
4      if len(lst) == 0:
5          return lst
6
7      pivot = lst[0]
8      pivots = [x for x in lst if x == pivot]
9      small = quicksort([x for x in lst if x < pivot])
10     large = quicksort([x for x in lst if x > pivot])
11
12     return small + pivots + large
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/20/quicksort.py>

# CALLABLES

- The emphasis in functional programming is on calling functions
- Python actually gives us several different ways to create functions, or at least something very *function-like* (i.e., that can be called):
  - 1 Regular functions created with `def` and given a name at definition time
  - 2 Anonymous functions created with `lambda`
  - 3 Generator functions
  - 4 *Closures* returned by function factories
    - ❑ Instances of classes that define a `__call__()` method
    - ❑ Static methods of instances, either via the `@staticmethod` decorator or via the class `__dict__`

# CALLABLES

- The emphasis in functional programming is on calling functions
- Python actually gives us several different ways to create functions, or at least something very *function-like* (i.e., that can be called):
  - 1 Regular functions created with `def` and given a name at definition time
  - 2 Anonymous functions created with `lambda`
  - 3 *Generator* functions
  - 4 *Closures* returned by function factories
  - 5 Instances of classes that define a `__call__` method
  - 6 Static methods of instances, either via the `@staticmethod` decorator or via the class `__dict__`

# NAMED FUNCTIONS AND LAMBDRAS (RECAP)

- The most obvious ways to create callables in Python are named functions and lambdas
- In most cases, lambda expressions are used within Python only for callbacks and other uses where a simple action is inlined into a function call

```
1  >>> def hello1(name):
2      .....     print("Hello", name)
3      .....
4  >>> hello2 = lambda name: print("Hello", name)
5  >>> hello1('John')
6  Hello John
7  >>> hello2('John')
8  Hello John
9
10 >>> hello3 = hello2 # can bind func to other names
11 >>> hello3.__qualname__
12 '<lambda>'
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/20/callable.py>

# CAVEATS, LIMITS, AND DISCIPLINE

## FUNCTIONAL PROGRAMMING STYLE

In most cases, one only leaks state intentionally, and creating a certain subset of all your functionality as pure functions allows for cleaner code

- One of the reasons that functions are useful is that they **isolate state lexically**
- This is a limited form of nonmutability in that (by default) nothing you do within a function will bind state variables outside the function
- This guarantee is very limited in that both the `global` and `nonlocal` statements explicitly allow state to “leak out” of a function
- Moreover, many **data types are themselves mutable**, so if they are passed into a function that function might change their contents
- Furthermore, **doing I/O can also change the “state of the world” and hence alter results of functions<sup>3</sup>**

<sup>3</sup>E.g., by changing the contents of a file or a database that is itself read elsewhere.

# CLOSURES AND CALLABLE INSTANCES

- A **closure** is “operations with data attached” (putting operations and data in the same object)
- Closures emphasise immutability and pure functions
- A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory

```
1  def make_adder(n):  
2      def adder(m):  
3          return m + n  
4      return adder  
5  
6  add5_f = make_adder(5)  # "functional"  
7  
8  >>> add5_f(10)  
9  15
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/20/closure.py>

# GENERATOR FUNCTIONS

- A special sort of function in Python is one that contains a `yield` statement, which turns it into a generator
- What is returned from calling such a function is not a regular value, but rather an `iterator` that produces a sequence of values as you call the `next()` function on it or loop over it
- For example, see the code for “Simple lazy [Sieve of Eratosthenes](#)” in `get_primes()`
- Every time you create a new object with `get_primes()` the iterator is the same infinite lazy sequence

⇒ [https://github.com/fpro-admin/lectures/blob/master/20/get\\_primes.py](https://github.com/fpro-admin/lectures/blob/master/20/get_primes.py)

# LAZY EVALUATION

## ITERATORS

Iterators are lazy sequences rather than realised collections

- Python does not quite offer *lazy data structures* in the sense of a language like Haskell
- However, use of the iterator protocol and Python's many built-in or standard library iteratables, accomplish much the same effect as an actual lazy data structure
- The easiest way to create an iterator — that is to say, a lazy sequence — in Python is to define a **generator function**
- Well, technically, the easiest way is to use one of the many *iterable objects* already produced by built-ins or the standard library rather than programming a custom one at all
- The module `itertools` is a collection of very powerful, and carefully designed, functions for performing *iterator algebra*

# HIGHER-ORDER FUNCTIONS

- Higher-order functions (often abbreviated as “HOF”) provide building blocks to express complex concepts by combining simpler functions into new functions
- In general, a higher-order function is simply a function that takes one or more functions as arguments and/or produces a function as a result
- It is common to think of `map()`, `filter()`, and `functools.reduce()` as the most basic building blocks of higher-order functions
- Almost as basic as map/filter/reduce as a building block is currying
  - In Python, currying is spelled as `partial()`, and is contained in the `functools` module
  - This is a function that will take another function, along with zero or more arguments to pre-fill, and return a function of fewer arguments that operates as the input function would when those arguments are passed to it

# EQUIVALENCIES

- The built-in functions `map()` and `filter()` are equivalent to comprehensions (especially now that generator comprehensions are available)

```
1 # Classic "FP-style"
2 transformed = map(transformation, iterator)
3
4 # Comprehension
5 transformed = (transformation(x) for x in iterator)
6
7 # Classic "FP-style"
8 filtered = filter(predicate, iterator)
9
10 # Comprehension
11 filtered = (x for x in iterator if predicate(x))
12
13 from functools import reduce
14 total = reduce(operator.add, it, 0)
15
16 # total = sum(it)
```

# EXERCISES

- Moodle activity at: [LE20: Effect-free programming style](#)

# PROGRAMMING FUNDAMENTALS

## FILES & PERSISTENCE

João Correia Lopes

INESC TEC, FEUP

11 december 2018

# CONTENTS

## 1 FILES

- 7.1 About files
- 7.2 Writing our first file
- 7.3 Reading a file line-at-a-time
- 7.4 Turning a file into a list of lines
- 7.5 Reading the whole file at once
- 7.6 An example
- 7.7 Directories
- 7.8 What about fetching something from the Web?

# FILES

MPFC:

**And now for something completely different!**

- Rather than avoiding side effects (effect-free programmings style)
- ... we focus on achieving persistence (doing I/O)

*“most **real computer programs** must retrieve stored information and record information for future use.”*

# PERSISTENCE

*"In computer science, persistence refers to the characteristic of **state that outlives the process that created it.***

*This is achieved in practice by storing the state as data in computer data storage. Programs have to transfer data to and from storage devices and have to **provide mappings** from the native programming-language data structures to the storage device data structures." [Wikipedia](#)*

# ABOUT FILES

- While a program is running, its data is stored in random access memory (RAM)
- RAM is fast and inexpensive, but it is also volatile
- To make data available the next time the program is started, it has to be written to a non-volatile storage medium
- Data on non-volatile storage media is stored in named locations called files

# FINDING A FILE ON YOUR DISK

- Opening a file requires that you, as a programmer, and Python agree about the location of the file on your disk
- The way that files are located on disk is by their **path**
- You can think of the **filename** as the short name for a file, and the **path** as the full name.

```
    └── reduce.py
    └── sort.py
    └── transpose.py
20
    ├── callables.py
    ├── closure.py
    ├── factorialR.py
    ├── get_primes.py
    └── quicksort.py
21
    └── HowtoThink3rd_2018.pdf
    └── README.md

21 directories, 121 files
jlopes@ubunas:~/GitProjects/fpro> █
```

# WRITING OUR FIRST FILE

- Opening a file creates what its called a **file handle**
- Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk
- Let's begin with a simple program that writes three lines of text into a file:

```
1  with open("test.txt", "w") as myfile:  
2      myfile.write("My first file written from Python\n")  
3      myfile.write("-----\n")  
4      myfile.write("Hello, world!\n")
```

- You may as well use: `f = open ("workfile", "w")`
- But, if you're not using the `with`, then you should call `f.close()` to close the file and immediately free up any system resources used by it

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/myfile.py>

# WRITING OUR FIRST FILE

- Opening a file creates what its called a **file handle**
- Our program calls methods on the handle, and this makes changes to the actual file which is usually located on our disk
- Let's begin with a simple program that writes three lines of text into a file:

```
1  with open("test.txt", "w") as myfile:  
2      myfile.write("My first file written from Python\n")  
3      myfile.write("-----\n")  
4      myfile.write("Hello, world!\n")
```

- You may as well use: `f = open ("workfile", "w")`
- But, if you're not using the `with`, then you should call `f.close()` to close the file and immediately free up any system resources used by it

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/myfile.py>

# MODES

- To manipulate files one needs to provide the path to the file and the mode for open

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

⇒ <https://docs.python.org/3/library/functions.html#open>

# READING A FILE LINE-AT-A-TIME

- Now that the file exists on our disk, we can open it, this time for reading, and read all the lines in the file, one at a time
- The for statement in line 2 reads everything up to and including the newline character

```
1  with open("test.txt", "r") as my_new_handle:  
2      for the_line in my_new_handle:  
3          # Do something with the line we just read.  
4          # Here we just print it.  
5          print(the_line, end="")
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/myfile.py>

# TURNING A FILE INTO A LIST OF LINES

- It is often useful to fetch data from a disk file and turn it into a list of lines
- The readlines method in line 2 reads all the lines and returns a list of the strings
  - We could read each line one-at-a-time and build up the list ourselves, but it is a lot easier to use the method that the Python implementors gave us!

```
1  with open("players.txt", "r") as input_file:  
2      all_lines = input_file.readlines()
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/players.py>

# READING THE WHOLE FILE AT ONCE

- Another way of working with text files is to read the complete contents of the file into a string, and then to use our string-processing skills to work with the contents
- By default, if we don't supply the mode, Python **opens the file for reading**

```
1  with open("somefile.txt") as f:  
2      content = f.read()  
3  
4      words = content.split()  
5      print("There are {0} words in the file.".format(len(words)))
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/players2.py>

# METHODS OF FILE OBJECTS

Method	Description
<code>f.read()</code>	reads the entire file
<code>f.readline()</code>	reads a single line from the file
<code>f.write(string)</code>	writes the contents of string to the file
<code>f.tell()</code>	returns an integer giving the file object's current position
<code>f.seek(offset, from)</code>	changes the file object's position

⇒ <https://docs.python.org/3.6/tutorial/inputoutput.html#methods-of-file-objects>

# A FILTER EXAMPLE

- Here is a filter that copies one file to another, omitting any lines that begin with #:

```
1 def filter(oldfile, newfile):
2     with open(oldfile, "r") as infile, open(newfile, "w") as outfile:
3
4         for line in infile:
5
6             # Put any processing logic here
7             if not line.startswith('#'):
8                 outfile.write(line)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/filter.py>

# DIRECTORIES

- Files on non-volatile storage media are organized by a set of rules known as a **file system**
- File systems are made up of **files** and **directories**, which are **containers** for both files and other **directories**
- When we open a file for reading, Python looks for it in the **current directory**
- If we want to open a file **somewhere else**, we have to **specify the path to the file**, which is the **name of the directory (or folder)** where the file is located

```
1  >>> wordsfile = open("/usr/share/dict/words", "r")
2  >>> wordlist = wordsfile.readlines()
3  >>> print(wordlist[:7])
4  ['A\n', "A's\n", 'AMD\n', "AMD's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

# FETCHING FROM THE WEB

- Here is a very simple example that copies the contents at some Web URL to a local file
- The `urlretrieve` function could be used to download any kind of content from the Web
- The resource we're trying to fetch must exist (check it using a browser)

```
1 import urllib.request  
2  
3 url = "https://www.ietf.org/rfc/rfc793.txt"  
4 destination_filename = "rfc793.txt"  
5  
6 urllib.request.urlretrieve(url, destination_filename)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/scraping.py>

# FETCHING FROM THE WEB USING REQUESTS

- The module `requests` is not part of the standard library
- It is easier to use and significantly more potent than the `urllib` module (see [docs](#))
- Read the web resource directly into a string and print that string

```
1 import requests  
2  
3 url = "https://www.ietf.org/rfc/rfc793.txt"  
4 response = requests.get(url)  
5 print(response.text)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/scraping.py>

## FURTHER READING

- Web Scraping in Python (using BeautifulSoup): [Beginner's guide](#)
- Data Persistence: [The Python Standard Library](#)
- DB-API 2.0 interface for SQLite databases: [The Python Standard Library](#)

⇒ <https://github.com/fpro-admin/lectures/blob/master/21/scraping.py>

# EXERCISES

- Moodle activity at: [LE21: Files and persistence](#)

# PROGRAMMING FUNDAMENTALS

## MODULES

João Correia Lopes

INESC TEC, FEUP

13 december 2018

# CONTENTS

## 1 MODULES

- 8.1 Random numbers
- 8.2 The `time` module
- 8.3 The `math` module
- 8.4 Creating your own modules
- 8.5 Namespaces
- 8.6 Scope and lookup rules
- 8.8 Three `import` statement variants

# MODULES

- A module is a file containing Python definitions and statements intended for use in other Python programs
- There are many Python modules that come with Python as part of the standard library
- We have seen some already: the `turtle` module, the `string` module, the `functools` module
- The help system contains a listing of all the standard modules that are available with Python
- Play with help!

⇒ <https://docs.python.org/3/library/>

# RANDOM NUMBERS

- We often want to use random numbers in programs
- Python provides a module random that helps with tasks like this
- The randrange method call generates an integer between its lower and upper argument, using the same semantics as range
- All the values have an equal probability of occurring (it's a *uniform distribution*)

```
1 import random
2
3 rng = random.Random() # create an object that generates random numbers
4
5 dice_throw = rng.randrange(1, 7) # Return one of 1,2,3,4,5,6
6 random_odd = rng.randrange(1, 100, 2)
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/random.py>

# REPEATABILITY AND TESTING

- Random number generators are based on a **deterministic** algorithm — repeatable and predictable
- So they're called **pseudo-random** generators — they are not genuinely random
- They start with a *seed* value
- Each time you ask for another random number, you'll get one based on the current seed attribute, and the state of the seed will be updated
- But, for debugging and for writing unit tests, it is convenient to have repeatability

1

```
drng = random.Random(123)    # generator with known starting state
```

# PICKING BALLS FROM BAGS, THROWING DICE, SHUFFLING A PACK OF CARDS

## ■ Pulling balls out of a bag with *replacement*

```
1 def make_random_ints(num, lower_bound, upper_bound):  
2     rng = random.Random()  
3     result = []  
4     for i in range(num):  
5         result.append(rng.randrange(lower_bound, upper_bound))  
6     return result
```

## ■ Pulling balls out of the bag *without replacement*

```
1 xs = list(range(1,13))    # Make list 1..12 (there are no duplicates)  
2 rng = random.Random()    # Make a random number generator  
3 rng.shuffle(xs)          # Shuffle the list  
4 result = xs[:5]           # Take the first five elements
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/22/random\\_ints.py](https://github.com/fpro-admin/lectures/blob/master/22/random_ints.py)

# PICKING BALLS FROM BAGS, THROWING DICE, SHUFFLING A PACK OF CARDS

## ■ Pulling balls out of a bag with *replacement*

```
1 def make_random_ints(num, lower_bound, upper_bound):  
2     rng = random.Random()  
3     result = []  
4     for i in range(num):  
5         result.append(rng.randrange(lower_bound, upper_bound))  
6     return result
```

## ■ Pulling balls out of the bag *without replacement*

```
1 xs = list(range(1,13))    # Make list 1..12 (there are no duplicates)  
2 rng = random.Random()    # Make a random number generator  
3 rng.shuffle(xs)          # Shuffle the list  
4 result = xs[:5]           # Take the first five elements
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/22/random\\_ints.py](https://github.com/fpro-admin/lectures/blob/master/22/random_ints.py)

# THE **TIME** MODULE

- The `time` module has a function called `clock` that can be used for *timing* programs
- Whenever `clock` is called, it returns a floating point number representing how many seconds have elapsed since your program started running

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/timing.py>

# THE **MATH** MODULE

- The `math` module contains the kinds of mathematical functions you'd typically find on your calculator
- Functions: `sin`, `cos`, `sqrt`, `asin`, `log`, `log10`
- Some mathematical constants like `pi` and `e`
- Angles are expressed in radians rather than degrees
- There are two functions `radians` and `degrees` to convert between these two popular ways of measuring angles
- Mathematical functions are “pure” and don't have any *state*

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/math.py>

# CREATING YOUR OWN MODULES

- All we need to do to create our own modules is to save our script as a file with a .py extension
- Suppose, for example, this script is saved as a file named `seqtools.py`

```
1 def remove_at(pos, seq):  
2     return seq[:pos] + seq[pos+1:]
```

- We can now use our module, both in scripts we write, or in the interactive Python interpreter
- To do so, we must first import the module

```
1 >>> import seqtools  
2 >>> s = "A string!"  
3 >>> seqtools.remove_at(4, s)  
4 'A sting!'
```

## \_\_NAME\_\_ (RECAP)

- Before the Python interpreter executes your program, it defines the variable `__name__`
  - The variable is automatically set to the string value "`__main__`" when the program is being executed by itself in a standalone fashion
  - On the other hand, if the program is being imported by another program, then the "`__name__`" variable is set to the name of that module
- This ability to conditionally execute our main function can be extremely useful when we are writing code that will potentially be used by others

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/mymath.py>

⇒ <https://github.com/fpro-admin/lectures/blob/master/09/import.py>

# NAMESPACES

- A **namespace** is a collection of identifiers that belong to a module, or to a function
- Each module has its own namespace, so we can use the same identifier name in multiple modules without causing an identification problem

```
1 # module1.py
2
3 question = "What is the meaning of Life, the Universe, and Everything?"
4 answer = 42
```

```
1 # module2.py
2
3 question = "What is your quest?"
4 answer = "To seek the holy grail."
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/namespaces.py>

# FUNCTION NAMESPACES

- Functions also have their own namespaces:

```
1  def f():
2      n = 7
3      print("printing n inside of f:", n)
4
5  def g():
6      n = 42
7      print("printing n inside of g:", n)
8
9  n = 11
10 f()
11 g()
```

*Python takes the module name from the file name, and this becomes the name of the namespace: `math.py` is a filename, the module is called `math`, and its namespace is `math`.*

# FUNCTION NAMESPACES

- Functions also have their own namespaces:

```
1  def f():
2      n = 7
3      print("printing n inside of f:", n)
4
5  def g():
6      n = 42
7      print("printing n inside of g:", n)
8
9  n = 11
10 f()
11 g()
```

*Python takes the module name from the file name, and this becomes the name of the namespace: `math.py` is a filename, the module is called `math`, and its namespace is `math`.*

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/fnamespaces.py>

# SCOPE AND LOOKUP RULES

- The **scope** of an identifier is the region of program code in which the identifier can be accessed, or used
- There are three important scopes in Python:
  - **Local scope** refers to identifiers declared within a function: these identifiers are kept in the namespace that belongs to the function, and each function has its own namespace
  - **Global scope** refers to all the identifiers declared within the current module, or file
  - **Built-in scope** refers to all the identifiers built into Python — those like `range` and `min` that can be used without having to import anything, and are (almost) always available
- Functions `locals`, `globals`, and `dir` to see what is the scope
- Python uses precedence rules: the innermost, or local scope, will always take precedence over the global scope, and the global scope always gets used in preference to the built-in scope

⇒ <https://github.com/fpro-admin/lectures/blob/master/22/scope.py>

# THREE IMPORT STATEMENT VARIANTS

- Here are three different ways to import names into the current namespace, and to use them:

```
1 # math is added to the current namespace
2 import math
3 x = math.sqrt(10)
4
5 # names are added directly to the current namespace
6 from math import cos, sin, sqrt
7 x = sqrt(10)
8
9 # import all the identifiers from math
10 from math import *
11 x = sqrt(10)
12
13 #
14 import math as m
15 m.pi
```

# EXERCISES

- Moodle activity at: [LE22: Modules](#)

# PROGRAMMING FUNDAMENTALS

## EXCEPTIONS

João Correia Lopes

INESC TEC, FEUP

18 december 2018

# CONTENTS

## 1 EXCEPTIONS

- E.1 Catching exceptions
- E.2 Raising our own exceptions
- E.3 Revisiting an earlier example
- E.4 The `finally` clause of the `try` statement
- The `assert` statement
- Examples & Summary

# SOME COMMON EXCEPTIONS

Here are some basic exceptions that you might encounter when writing programs:

- `NameError` — raised when the program cannot find a local or global name
- `TypeError` — raised when a function is passed an object of the inappropriate type as its argument
- `ValueError` — occurs when a function argument has the right type but an inappropriate value
- `ZeroDivisionError` — raised when you provide the second argument for a division or modulo operation as zero
- `FileNotFoundException` — raised when the file or directory that the program requested does not exist

⇒ <https://code.tutsplus.com/tutorials/>

# RUNTIME ERRORS

- Whenever a runtime error occurs, it creates an exception object
- The program stops running at this point and Python prints out the traceback, which ends with a message describing the exception that occurred
- The error message on the last line has two parts: the type of error before the colon, and specifics about the error after the colon

```
1      >>> tup = ("a", "b", "d", "d")
2      >>> tup[2] = "c"
3      Traceback (most recent call last):
4          File "<interactive input>", line 1, in <module>
5      TypeError: 'tuple' object does not support item assignment
```

# CATCHING EXCEPTIONS

- Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop
- We can handle the exception using the `try` statement to "wrap" a region of code

```
1 filename = input("Enter a file name: ")
2 try:
3     f = open(filename, "r")
4 except FileNotFoundError:
5     print("There is no file named", filename)
```

- A `else` block is executed after the `try` one, if no exception occurred
- A `finally` block is executed in any case

# CATCHING EXCEPTIONS

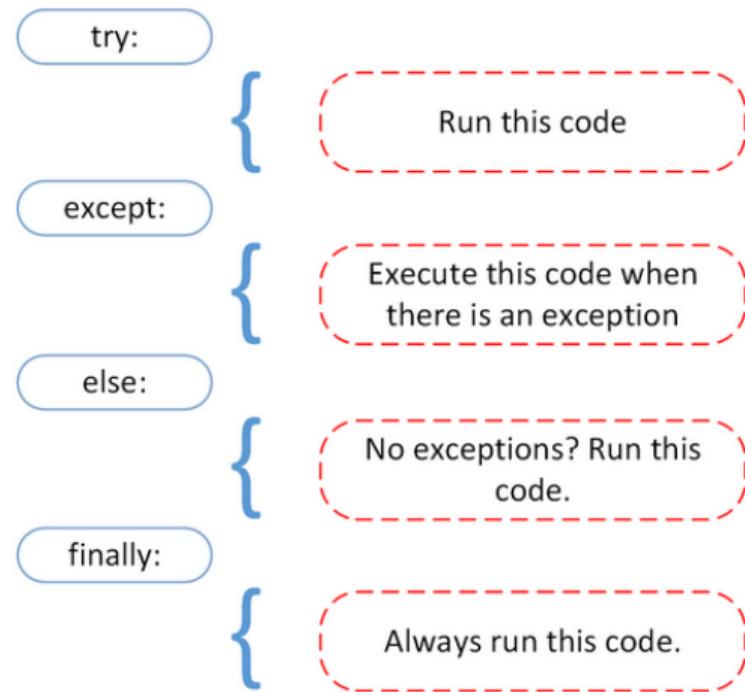
- Sometimes we want to execute an operation that might cause an exception, but we don't want the program to stop
- We can handle the exception using the `try` statement to "wrap" a region of code

```
1 filename = input("Enter a file name: ")
2 try:
3     f = open(filename, "r")
4 except FileNotFoundError:
5     print("There is no file named", filename)
```

- A `else` block is executed after the `try` one, if no exception occurred
- A `finally` block is executed in any case

⇒ <https://github.com/fpro-admin/lectures/blob/master/23/try.py>

# CATCHING EXCEPTIONS



⇒ <https://realpython.com/python-exceptions/>

# A COMPLETE EXAMPLE

```
1 import math
2
3 number_list = [10, -5, 1.2, 'apple']
4
5 for number in number_list:
6     try:
7         number_factorial = math.factorial(number)
8     except TypeError:
9         print("Factorial is not supported for given input type.")
10    except ValueError:
11        print("Factorial only accepts positive integer values.", number, " is not a
12            positive integer.")
13    else:
14        print("The factorial of", number, "is", number_factorial)
15    finally:
16        print("Release any resources in use.")
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/23/example.py>

# RAISING OUR OWN EXCEPTIONS

- Can our program deliberately cause its own exceptions?
- If our program detects an error condition, we can raise an exception
- If there's a chain of calls, "*unwinding the call stack*" takes place until a `try ... except` is found

```
1 def get_age():
2     age = int(input("Please enter your age: "))
3     if age < 0:
4         # Create a new instance of an exception
5         my_error = ValueError("{0} is not a valid age".format(age))
6         raise my_error
7     return age
```

⇒ <https://github.com/fpro-admin/lectures/blob/master/23/age.py>

# REVISITING AN EARLIER EXAMPLE

- Using exception handling, we can now modify our `recursion_depth` example from the previous chapter so that it stops when it reaches the maximum recursion depth allowed

⇒ [https://github.com/fpro-admin/lectures/blob/master/23/rec\\_depth.py](https://github.com/fpro-admin/lectures/blob/master/23/rec_depth.py)

## FINALLY

- A common programming pattern is to grab a resource of some kind
- Then we perform some computation which may raise an exception, or may work without any problems
- Whatever happens, we want to “clean up” the resources we grabbed

⇒ [https://github.com/fpro-admin/lectures/blob/master/23/show\\_poly.py](https://github.com/fpro-admin/lectures/blob/master/23/show_poly.py)

# ASSERTIONS

- Assertions are statements that assert or state a fact
- Assertions are simply boolean expressions that checks if the conditions return true or not: if it's false, the program stops and throws an error
- assert statement takes an expression and optional message
- Assertions are used to check types, values of argument and the output of the function
- Assertions are used as debugging tool as it halts the program at the point where an error occurs

⇒ <https://github.com/fpro-admin/lectures/blob/master/23/assert.py>

# THE MOST DIABOLICAL PYTHON ANTI PATTERN

- There are plenty of ways to write bad code. But in Python, one in particular reigns as king

```
1  try:  
2      do_something()  
3  except:  
4      pass
```

⇒ <https://realpython.com/the-most-diabolical-python-antipattern/>

# VALIDATE USER INPUT

```
1  def inputNumber(message):
2      while True:
3          try:
4              userInput = int(input(message))
5          except ValueError:
6              print("Not an integer! Try again.")
7              continue
8          else:
9              return userInput
10
11 age = inputNumber("How old are you?")
```

# NESTED TRY

```
1  try:
2      try:
3          raise ValueError('1')
4      except TypeError:
5          print("Caught the type error")
6  except ValueError:
7      print("Caught the value error!")
```

⇒ [https://github.com/fpro-admin/lectures/blob/master/23/nested\\_try.py](https://github.com/fpro-admin/lectures/blob/master/23/nested_try.py)

# SUMMING UP

- After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python:
  - `raise` allows you to throw an exception at any time
  - `assert` enables you to verify if a certain condition is met and throw an exception if it isn't
  - In the `try` clause, all statements are executed until an exception is encountered
  - `except` is used to catch and handle the exception(s) that are encountered in the `try` clause
  - `else` lets you code sections that should run only when no exceptions are encountered in the `try` clause
  - `finally` enables you to execute sections of code that should always run, with or without any previously encountered exceptions

# EXERCISES

- Moodle activity at: [LE23: Exceptions](#)

# PROGRAMMING FUNDAMENTALS

## DEBUGGING

João Correia Lopes

INESC TEC, FEUP

20 december 2018

# CONTENTS

## 1 DEBUGGING

- Introduction
- Debugging Syntax errors
- Debugging Runtime errors
- Debugging Semantic errors

## 2 HOW TO AVOID DEBUGGING (RECAP)

## 3 DEBUGGING IN SPYDER3

## KINDS OF ERRORS

Different *kinds of errors* can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- 1 **Syntax errors** are produced by Python when it is translating the source code into byte code
- 2 **Runtime errors** are produced by the runtime system if something goes wrong while the program is running
- 3 **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing

# KINDS OF ERRORS (DETAILS)

1 **Syntax errors** are produced by Python when it is translating the source code into byte code

- They usually indicate that there is something wrong with the syntax of the program
- Example: Omitting the colon at the end of a def statement yields the somewhat redundant message  
`SyntaxError: invalid syntax`

2 **Runtime errors** are produced by the runtime system if something goes wrong while the program is running

- Most runtime error messages include information about where the error occurred and what functions were executing
- Example: An infinite recursion eventually causes a runtime error of maximum recursion depth exceeded

3 **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing

- Example: An expression may not be evaluated in the order you expect, yielding an unexpected result

# KINDS OF ERRORS (DETAILS)

- 1 **Syntax errors** are produced by Python when it is translating the source code into byte code
  - They usually indicate that there is something wrong with the syntax of the program
  - Example: Omitting the colon at the end of a def statement yields the somewhat redundant message  
`SyntaxError: invalid syntax`
- 2 **Runtime errors** are produced by the runtime system if something goes wrong while the program is running
  - Most runtime error messages include information about where the error occurred and what functions were executing
  - Example: An infinite recursion eventually causes a runtime error of maximum recursion depth exceeded
- 3 **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing
  - Example: An expression may not be evaluated in the order you expect, yielding an unexpected result

# KINDS OF ERRORS (DETAILS)

- 1 **Syntax errors** are produced by Python when it is translating the source code into byte code
  - They usually indicate that there is something wrong with the syntax of the program
  - Example: Omitting the colon at the end of a def statement yields the somewhat redundant message  
`SyntaxError: invalid syntax`
- 2 **Runtime errors** are produced by the runtime system if something goes wrong while the program is running
  - Most runtime error messages include information about where the error occurred and what functions were executing
  - Example: An infinite recursion eventually causes a runtime error of maximum recursion depth exceeded
- 3 **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing
  - Example: An expression may not be evaluated in the order you expect, yielding an unexpected result

# FIRST STEP IN DEBUGGING

- The first step in debugging is to figure out which kind of error you are dealing with
- Although the following sections are organized by error type, some techniques are applicable in more than one situation

# SYNTAX ERRORS

- Syntax errors are usually easy to fix once you figure out what they are
- Here are some ways to avoid the most common syntax errors ([RLE](#))
- I can't get my program to run no matter what I do ([RLE](#))

# RUNTIME ERRORS

- Once your program is syntactically correct, Python can import it and at least start running it
- What could possibly go wrong?
  - My program does absolutely nothing ([RLE](#))
  - My program hangs ([RLE](#))
  - Infinite Loop ([RLE](#))
  - Infinite Recursion ([RLE](#))
  - Flow of Execution ([RLE](#))
  - When I run the program I get an exception ([RLE](#))
  - I added so many `print` statements I get inundated with output ([RLE](#))

# SEMANTIC ERRORS

- In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong
- Only you know what the program is supposed to do, and only you know that it isn't doing it
- The first step is to make a connection between the program text and the behavior you are seeing
  - My program doesn't work ([RLE](#))
  - I've got a big hairy expression and it doesn't do what I expect ([RLE](#))
  - I've got a function or method that doesn't return what I expect ([RLE](#))
  - I'm really, really stuck and I need help ([RLE](#))
  - No, I really need help ([RLE](#))
- The time it takes to insert a few well-placed print statements is often short compared to setting up the debugger, inserting and removing breakpoints, and walking the program to where the error is occurring.
- Do not forget you may use assert!

# SEMANTIC ERRORS

- In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong
- Only you know what the program is supposed to do, and only you know that it isn't doing it
- The first step is to make a connection between the program text and the behavior you are seeing
  - My program doesn't work ([RLE](#))
  - I've got a big hairy expression and it doesn't do what I expect ([RLE](#))
  - I've got a function or method that doesn't return what I expect ([RLE](#))
  - I'm really, really stuck and I need help ([RLE](#))
  - No, I really need help ([RLE](#))
- The time it takes to insert a few well-placed `print` statements is often short compared to setting up the debugger, inserting and removing breakpoints, and walking the program to where the error is occurring.
- Do not forget you may use `assert`!

# HOW TO BE A SUCCESSFUL PROGRAMMER (RECAP)

- One of the most important skills you need to acquire is the ability to debug your programs
- Debugging is a skill that you need to master over time
- As programmers we spend 99% of our time trying to get our program to work
- But here is the secret, when you are successful, you are happy, your brain releases a bit of chemical that makes you feel good
- **Start small, get something small working, and then add to it**

# HOW TO AVOID DEBUGGING (RECAP)

## MANTRA

Get something working and keep it working

- **Start Small**

- This is probably the single biggest piece of advice for programmers at every level.

- **Keep it working**

- Once you have a small part of your program working the next step is to figure out something small to add to it.

Ok, let's look at an example: the `alarm_clock.py` of RE02

⇒ [https://github.com/fpro-admin/lectures/blob/master/06/alarm\\_clock.py](https://github.com/fpro-admin/lectures/blob/master/06/alarm_clock.py)

# BEGINNING TIPS FOR DEBUGGING (RECAP)

Debugging a program is a different way of thinking than writing a program.  
The process of debugging is much more like being a detective.

## 1 Everyone is a suspect (Except Python)!

### Find clues

- Error messages
- Print statements

# BEGINNING TIPS FOR DEBUGGING (RECAP)

Debugging a program is a different way of thinking than writing a program.  
The process of debugging is much more like being a detective.

- 1 Everyone is a suspect (Except Python)!
- 2 Find clues
  - Error messages
  - Print statements

# SUMMARY ON DEBUGGING (RECAP)

- Make sure you take the time to understand error messages
  - They can help you a lot
- print statements are your friends
  - Use them to help you uncover what is **really** happening in your code
- Work backward from the error
  - Many times an error message is caused by something that has happened before it in the program
  - Always remember that Python evaluates a program top to bottom

# DEBUGGING IN SPYDER

- Debugging in Spyder is supported through integration with the enhanced *ipdb debugger* in the IPython Console ([Spyder | Docs](#))
- This allows **breakpoints** and the execution flow to be viewed and controlled right from the Spyder GUI, as well as with all the familiar IPython console commands
- Inspecting variables ([Spyder | Docs](#))

⇒ <https://github.com/fpro-admin/lectures/blob/master/23/variables.py>

# MONTY PYTHON'S LIFE OF BRIAN

## Always Look On The Bright Side Of Life (Lyrics)

*Some things in life are bad they can really make you mad*

*Other things just make you swear and curse*

*When you're chewin' on life's gristle, don't grumble give a whistle*

*And this'll help things turn out for the best*

*And always look on the bright side of life*

*Always look on the light side of life*

*If life seems jolly rotten there's something you've forgotten*

*And that's to laugh and smile and dance and sing*

...

⇒ [https://www.youtube.com/watch?v=Ep9Vzb6R\\_58](https://www.youtube.com/watch?v=Ep9Vzb6R_58)

# EXERCISES

- Moodle activity at: [LE24: Debugging](#)