Computer Code for Beginners
Week 2

Matt Luckcuck

28/09/2017

Course Outline

Course Topics

- Introduce Programming and Python
- More Loops, Lists and Basic Functions
- More Sequences and Functions
- More Complex Data Types
- Handling Errors, File Handling
- Larger Two-Module Program

Housekeeping

Housekeeping

- Toilets
- Fire Alarm
- Additional Support

Housekeeping

Absences and Information

Fulford School

- Tel: 01904 611 505
- Email: fulford.adulted@york.gov.uk

Matt Luckcuck (Me)

- Email: m.luckcuck@gmail.com

Housekeeping

Expectations

- Keeping notes and language cheat-sheet
  - Paper or Electronic
  - Pages of notes
  - Flash cards
  - Spider diagram
  - Your own presentation slides
- Independently searching for some information
  - Online is fine
- **Please** ask questions if you're unsure!

Housekeeping

Each Week...

- $\sim$ 40 minutes of lecture
- $\sim$ 15 minutes break
- $\sim$ 65 minutes of practical

Last Time

Previously...

- Computers are **stupid**
- Introduction to...
  - Variables
  - Sequential Instructions
  - Branching
  - Basic Loops

This Week

Outline

- Recap
- Decomposition
- More Loops
- Lists
- Functions
- Exercises Overview

Recap

Recap

Modules

- A *module* is a collection of code that performs a function
- In Python, each file is a module
- Design decision...
  - A simple program is likely to be one module
  - A more complex program is best split up into separate modules

Recap

Functions

- A block of code, wrapped up for us to use when we need it
- Lots of built-in functions (like `print()`)
- We can write our own
- Can take parameters (like `print("Hello World")`)
- Can return values
- Proper introduction to these later

Recap

Variable

- Data that our program uses
- Box in the computer's memory with a value inside
- Label to remember what's inside
- `name = value`
  - Assigning a `value` to the `name`
- So naming variables well is important!

Recap

Variable Types

- Whole Numbers — Integers (eg 1 or 10)
- Decimal Numbers — Floating Point Numbers (eg 3.14)
- Boolean (`True` or `False`)
- String of characters (Text)
- Others...

Recap

Boolean Operators

- `not x`
  - Negates (toggles) the value
- `x and y`
  - `True` if both values are `True`
- `x or y`
  - `True` if at least one value is `True`

Recap

Strings

- String is a sequence of characters
  - Either `"Hello World"` or `'Hello World'`
- A character is represented 'internally' by a unique code
  - UniCode
- We can convert between characters and their code
- `ord('a')` – 97
- `chr(97)` – 'a'

Recap

Sequential Instructions

- A program is a sequence of instructions. . .
  - Unless we tell it otherwise
- Sequential instructions are a basic building block
  - But often too simple

Recap

Branching Control Structure

- Choice between one branch or another branch
  - Based on a boolean condition

```
if <condition>:
  <if block>
else:
  <else block>
```

  - Two blocks that are executed *conditionally*
  - Blocks must be indented

Recap

Looping Instructions

- Allows us to repeat a block of code
  - Iteration

```
while <condition>:
  <body>
```

  - Checks the condition at the being of each iteration
  - Executes the body of the loop *while* that condition is true
  - Loop body must be indented
- Need to be careful of infinite loops!

Recap

Python User Input

- In Python provides the `input()` function
  - `result = input("Type Something Please")`

Python Programming Style

Programming Style

- Python groups blocks of code by how indented they are
  - Can be tabs *or* spaces...
  - Pick one and stick to it

Recap

Good Practice

- Code Comments
  - `# A single-line comment`
  - Good for describing complicated code
  - Not an excuse for poor naming!
  - Also useful for temporarily removing a line
- Documentation
  - String on first line of a module or function
  - `""" describe what it does """`
  - Again, not an excuse for bad naming!
- Useful for people reading your code in the future
  - Which could be future-you!

Decomposition

Decomposition

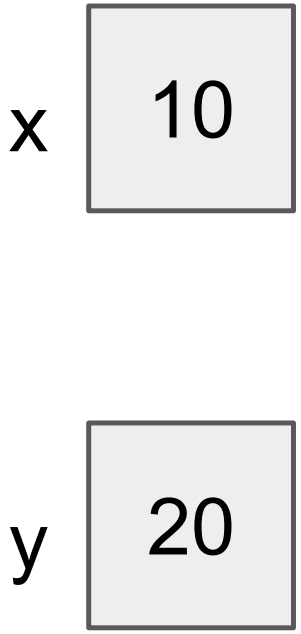
Values and Expressions

- The variable swap exercise shows us two ways of assigning variables
  - Literal value (`x = 10`)
  - Expressions (`temp = x`)
- Literal values assign the number (or string, etc) to the variable
- Expressions are *evaluated* to get their value
- Evaluating a variable name gives us it's value
  - In `temp = x` we get the value of `x` and store it in `temp`
  - The same happens if we use (e.g.) `result = input("...")`
- This example also introduces *decomposition*...

Decomposition

Decomposition Example

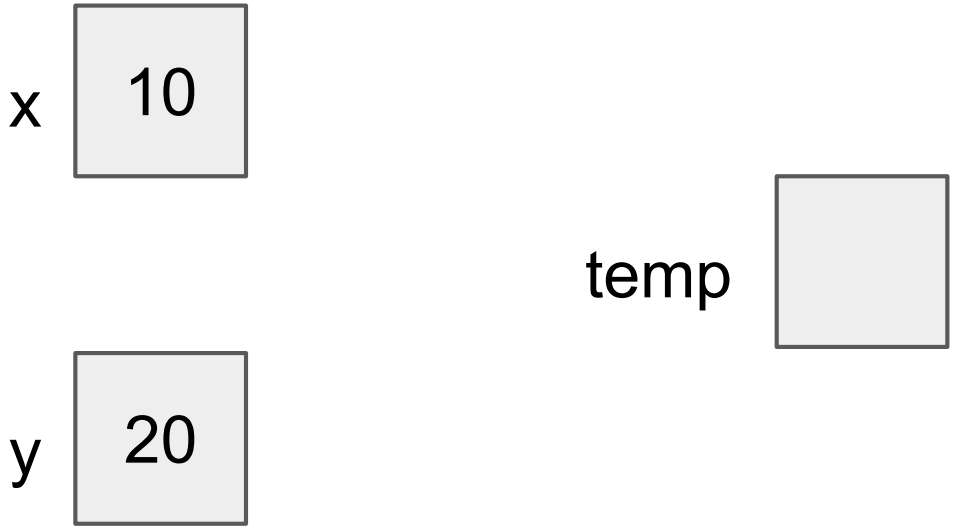
- Variables are a box in memory
- We want to swap the contents of these boxes. . .
- We need to break it down into steps



Decomposition

Decomposition Example

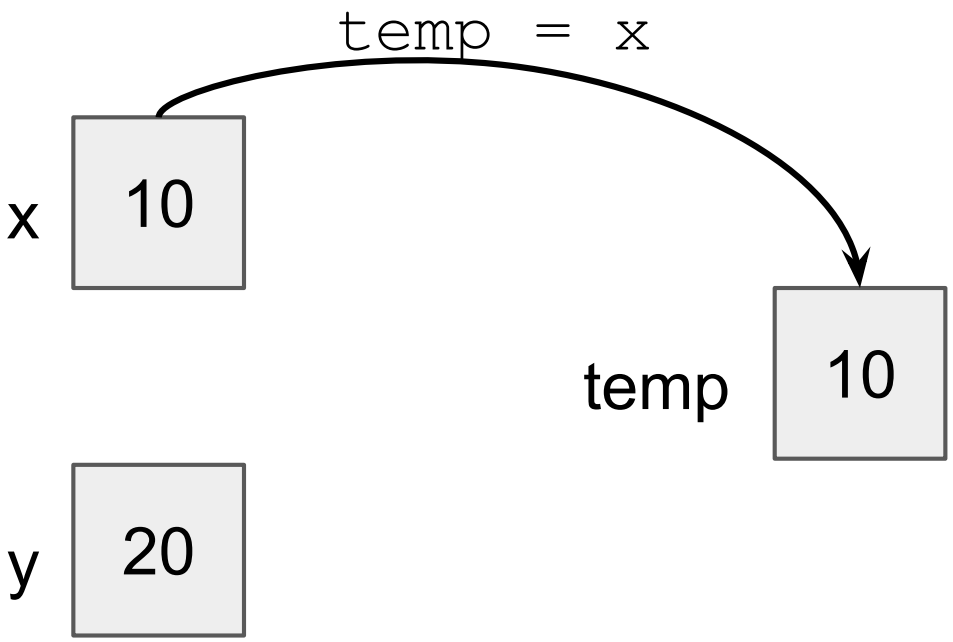
- All we can do with `x` and `y` is overwrite them
- So we make a new temporary variable



Decomposition

Decomposition Example

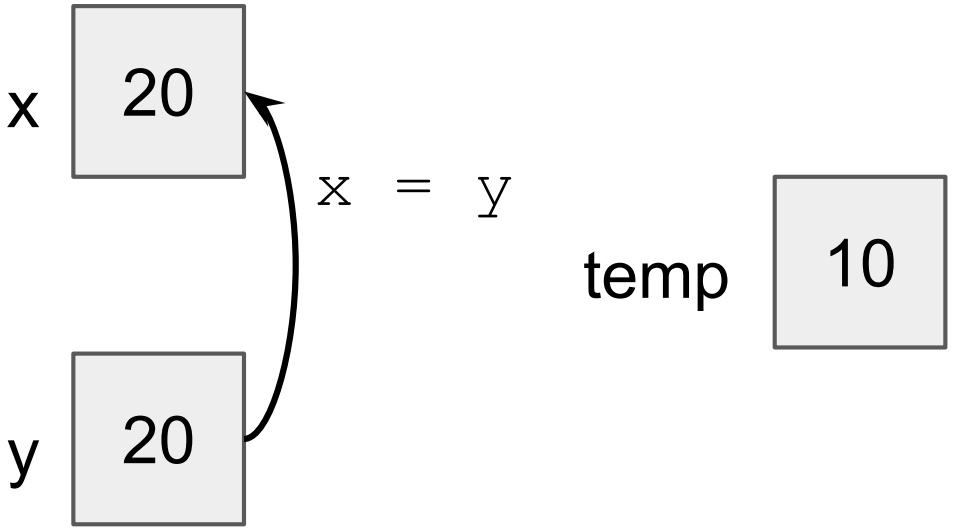
- We can 'overwrite' `temp`
- Banks the value of `x`



Decomposition

Decomposition Example

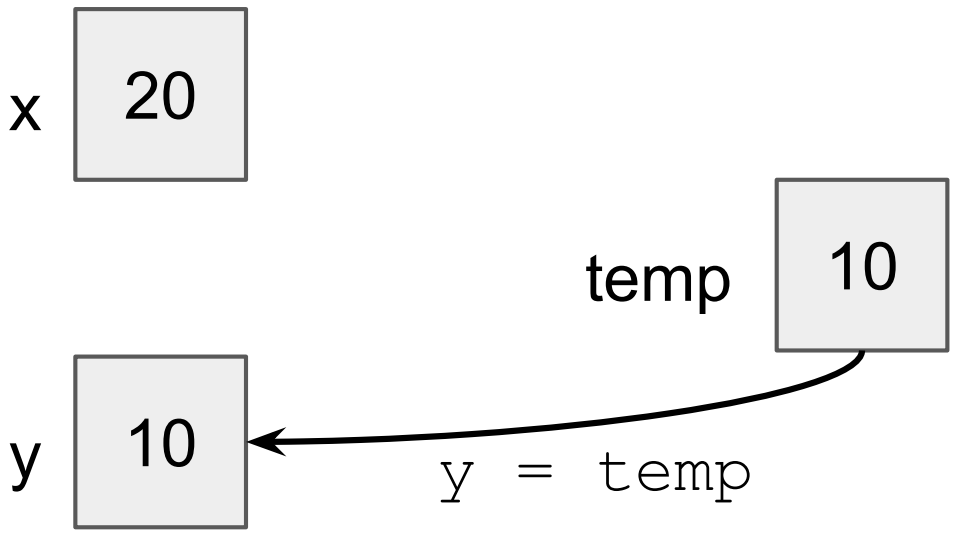
- Then we can overwrite `x`
- Note the value of `x` is still safe in `temp`



Decomposition

Decomposition Example

- Overwrite `y`
- Using `temp`



Decomposition

Another Decomposition Example

- Logo Shapes
- We know how to draw a triangle
- But the computer needs each step explained to it
  - Forward...
  - Turning...

Decomposition

Decomposition Examples

- Example of step-by-step instructions that computers need
- Computer is a tool
- Don't run the program in your head, make the computer do the work
- Learning how to do this takes some time

(More) Loops and Lists

Looping

Iteration (Looping)

- We looked at looping last week

```
while <condition>:
  <body>
```

- This basic version loops *while* `b == True`
- Python (and other languages) have another type of loop, the *for* loop

Looping

For Loops

- Repeat code *for* a certain number of times...
- Python does this by looping over a sequence
  - Simplest example is a list: `[1,2,3]`

```
for i in theList:
  <body>
```

- Loops for each item in `theList`
  - Or *while* there are more items in the list
- For each `item` in the list, we run the `<body>`
- `i` (the *loop index*) is incremented each iteration
  - Each iteration, `i` points at the next item

Lists

Lists Intro

- Compound data type
  - Sequence
- Simple ordered sequence of items
  - `numbers = [3,7,2]`
  - `colours = ["Red", "Blue", "Green"]`
- Zero-Indexed by sequential numbers
  - `[0↦"Red", 1↦"Blue", 2↦ "Green"]`
  - Highest index is $length - 1$

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?
    - `3`

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?
    - `3`
  - What about `colours[1]`?

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?
    - `3`
  - What about `colours[1]`?
    - `Blue`

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?
    - `3`
  - What about `colours[1]`?
    - `Blue`
  - What about `colours[3]`?

Lists

List Index

- Access item using index
  - `aList[0]` is first item in `aList`
- `numbers = [3,7,2]`
- `colours = ["Red", "Blue", "Green"]`
  - E.g. `numbers[0]` is?
    - `3`
  - What about `colours[1]`?
    - `Blue`
  - What about `colours[3]`?
    - `IndexError: list index out of range`

Lists

Useful List Operations

- `theList = [ ]` – makes a new empty list
- `theList[3] = 10` – *updates* index 3 to 10
- `len(theList)` – gives the length of `theList`
  - Note that the length is one more than the highest index
  - `len()` works for other data types too
- `x in theList` – `True` if `x` is in `theList`
- `theList.append(10)` – adds 10 to the end of `thelist`
- `theList.remove(10)` – removes the first 10 in `theList`

Looping

For Loops

- If we know the number of times we want to repeat our loop...
  - We can use the `range(x)` function
  - Returns a sequence of numbers `x` items long

```
for i in range(10):
  print(i)
```

Looping

For Loops

- If we know the number of times we want to repeat our loop...
  - We can use the `range(x)` function
  - Returns a sequence of numbers `x` items long
  - Be careful about the actual numbers

```
for i in range(10):
  print(i)
```

Prints 0 – 9

Looping

For Loops

- If we want to control the loop index...
  - `range(x, y)` counts from x upto (but not including) y

```
for i in range(1, 11):
  print(i)
```

Prints 1 – 10

Looping

For Loops

- If we want to look at every item in a list...
  - Use the same form but replace the `range()` call with our list

```
colours = ["Red", "Blue", "Green"]
for item in colours:
  print(item)
```

Looping

For Loops

- If we want to look at every item in a list...
  - Use the same form but replace the `range()` call with our list

```
colours = ["Red", "Blue", "Green"]
for item in colours:
  print(item)
```

Prints Red, Blue, and Green

Functions

Functions

Function

- Block of code wrapped up that does something for us
  - Function defined with: `def funcName():`
  - Function called using `funcName()`
- Function body is an indented block
- Naming is important
- Documentation string
  - `""" Describes what the function does """`
- We've seen some built-in functions:
  - `print()`
  - `len()`
  - `range()`

Functions

Function

- We can pass data into a function...
  - Called *parameters*
- Functions can *read* variables defined outside
  - More one this next week...
- Function may pass us back some data...
  - Called the *return value*
  - Imagine the return value replacing the call to the function
- Functions with no `return` statement return `None`
  - `None` is a type that represents nothing

Functions

Function Call Example

- Simple function to add two numbers
- `return`s the sum of `a` and `b`
  - `a` and `b` are whatever numbers we pass into the function
  - Function calls are expressions, so they're *evaluated*

```
1 def add(a, b):
2   """ Adds a to b """
3   return a+b
4
5 result = add(2,2)
```

Functions

Function Call Example

- Simple function to add two numbers
- `return`s the sum of `a` and `b`
  - `a` and `b` are whatever numbers we pass into the function
  - Function calls are expressions, so they're *evaluated*

```
def add(2, 2):
  """ Adds a to b """
  return a+b

result = add(2,2)
```

Functions

Function Call Example

- Simple function to add two numbers
- `return`s the sum of `a` and `b`
  - `a` and `b` are whatever numbers we pass into the function
  - Function calls are expressions, so they're *evaluated*

```
def add(2, 2):
  """ Adds a to b """
  return 2+2

result = add(2,2)
```

Functions

Function Call Example

- Simple function to add two numbers
- `return`s the sum of `a` and `b`
  - `a` and `b` are whatever numbers we pass into the function
  - Function calls are expressions, so they're *evaluated*

```
def add(2, 2):
  """ Adds a to b """
  return 4

result = add(2,2)
```

Functions

Function Call Example

- Simple function to add two numbers
- `return`s the sum of `a` and `b`
  - `a` and `b` are whatever numbers we pass into the function
  - Function calls are expressions, so they're *evaluated*

```
def add(2, 2):
  """ Adds a to b """
  return 4

result = 4
```

Functions

Why?

- Code reuse
- Simplifying the main program
- Single point of change

Summary

Summary

Summary

- For loops
  - `for i in range(10):`
  - `for i in aList:`
- Lists
  - Ordered sequences of values
  - Zero-indexed by numbers
- Functions
  - Wrapping up a block of code
  - `def name():`
  - Naming is important!

Summary

Practicals

- Logo Shapes2 (with Loops)
- Day of the Week Lists
- Random Number Guessing Game
- Caesar Cipher
- Course Website: `mluckcuck.github.io/python/`
- Manual: `docs.python.org/3/library/`
  - Make sure you use **Version 3**!