Programming with Python

Dr Ben Dudson

Department of Physics, University of York

21st January 2011

http://www-users.york.ac.uk/∼bd512/teaching.shtml

From last time...

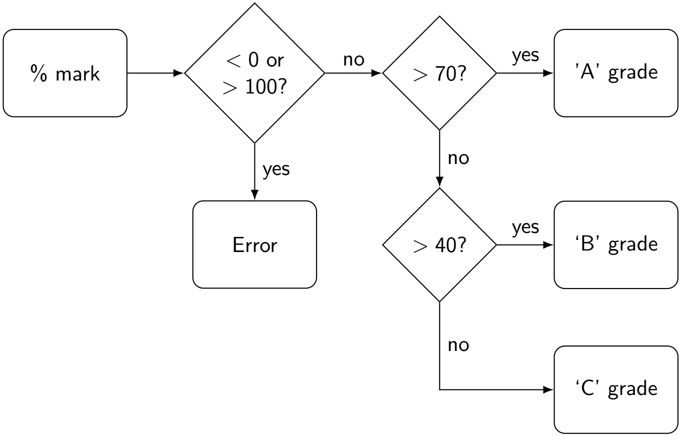
Last time we looked at problem solving, and methods for thinking up algorithms

- Understanding a problem
- Breaking it up into smaller problems
- Working out a set of steps to follow (recipe / algorithm)
- Using flow diagrams to express algorithms

This lecture, we'll start looking at programming, and the Python language

Programming

- Last lecture we looked at problem solving, and writing algorithms (instructions) as flow charts.



Programming

- Last lecture we looked at problem solving, and writing algorithms (instructions) as flow charts.
- The actual computer hardware (CPU) needs instructions in numerical codes which are very hard for humans to write, and even harder to read.

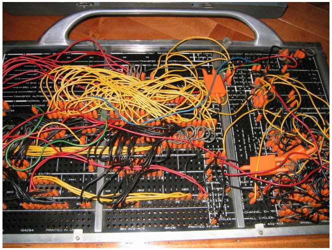
```
83 C0 01 A3 88 03 06 08 FF 14 85 90
```

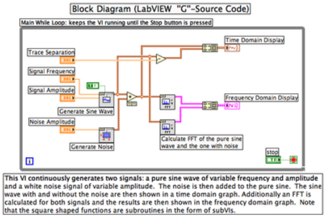
Programming

- Last lecture we looked at problem solving, and writing algorithms (instructions) as flow charts.
- The actual computer hardware (CPU) needs instructions in numerical codes which are very hard for humans to write, and even harder to read.

83 C0 01 A3 88 03 06 08 FF 14 85 90

- Instead, people have designed many different ways to express algorithms which can be understood by the computer




Block Diagram (LabVIEW "G"-Source Code)
Main While Loop: keeps the VI running until the Stop button is pressed
Trace Separation
Signal Frequency
Signal Amplitude
Generate Sine Wave
Noise Amplitude
Generate Noise
Time Domain Display
Frequency Domain Display
Calculate FFT of the pure sine wave and the one with noise
stop

This VI continuously generates two signals: a pure sine wave of variable frequency and amplitude and a white noise signal of variable amplitude. The noise is then added to the pure sine. The sine wave with and without the noise are then shown in a time domain graph. Additionally an FFT is calculated for both signals and the results are then shown in the frequency domain graph. Note that the square shaped functions are subroutines in the form of subVIs.

Programming

- Last lecture we looked at problem solving, and writing algorithms (instructions) as flow charts.
- The actual computer hardware (CPU) needs instructions in numerical codes which are very hard for humans to write, and even harder to read.

  83 C0 01 A3 88 03 06 08 FF 14 85 90

- Instead, people have designed many different ways to express algorithms which can be understood by the computer
- The best way to explain algorithms to computers has been found to be writing commands in plain text
- Languages which are readable to humans, but can be automatically converted into machine instructions
- Many many different ones around. Evolve over time, but use many of the same ideas

Python programming language

- Python is a relatively new language (1991, major update 2008)
- Intended to be clear and easy to start using, but at the same time very powerful for experienced users
- Widely used as it allows quicker development compared to more traditional languages (e.g. C or Fortran)
- Becoming quite common in scientific programming
- Freely available, open source. You can download and install a copy at home, on your laptop etc.

Variables

We saw in the last lecture that we give quantities names like “x”, “A” etc. These are like variables (x, y, z) in mathematics, but with one crucial difference

In maths, once you write $x = 2$, the value is fixed. In (most) programming languages, the value of variables can be changed

Variables

We saw in the last lecture that we give quantities names like “x”, “A” etc. These are like variables (x, y, z) in mathematics, but with one crucial difference

In maths, once you write $x = 2$, the value is fixed. In (most) programming languages, the value of variables can be changed

This is why it makes sense in Python to write

```
i = i + 1
```

In mathematics this makes no sense, but in programming what it means is “Calculate `i + 1`, then set `i` to this new value”, or if you prefer “Set `i` to the old `i` + 1”

$$i \to i + 1$$

Variable names

- Variable names in programs can be just single characters like `x` or `i`, but usually it's better to make them more descriptive
- This helps make your program easier to understand: it's easier to guess what `height` represents than `h`.

Names must start with a letter or underscore "_", and be a combination of letters, numbers, and underscores

Variable names

- Variable names in programs can be just single characters like `x` or `i`, but usually it's better to make them more descriptive
- This helps make your program easier to understand: it's easier to guess what `height` represents than `h`.

Names must start with a letter or underscore "_", and be a combination of letters, numbers, and underscores

Exercise: Which of these can be used as variable names?

1. position
2. 2ndValue
3. mark%
4. mark_percentage
5. my−variable
6. studentMark

Variable names

- Variable names in programs can be just single characters like `x` or `i`, but usually it's better to make them more descriptive
- This helps make your program easier to understand: it's easier to guess what `height` represents than `h`.

Names must start with a letter or underscore "_", and be a combination of letters, numbers, and underscores

Exercise: Which of these can be used as variable names?

1. position **Yes**
2. 2ndValue **No** - starts with a number
3. mark% **No** - not allowed % symbol
4. mark_percentage **Yes**
5. my−variable **No** - Not allowed minus symbol
6. studentMark **Yes**

Variable names

Some words have special meaning in Python, so can't be used as variable names.

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

Variable names

Some words have special meaning in Python, so can't be used as variable names.

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

Another important thing to remember is

Python is **case sensitive**, meaning that it treats upper case (A,B,C,...) as different characters to lower case (a,b,c,...)

This applies to all names in Python, so that these variables are all different

- myVariable
- MyVariable
- myvariable
- mYVaRIaBLe

Calculations

The basic operations in Python are

Power	A ** B	A^B
Multiply	A * B	$A \times B$
Divide	A / B	A/B
Add	A + B	$A + B$
Subtract	A − B	$A - B$

Operations higher up the list (e.g. power) are done before those lower down (e.g. add). This is called operator **precedence**.

Calculations

The basic operations in Python are

Power	A ** B	A^B
Multiply	A * B	$A \times B$
Divide	A / B	A/B
Add	A + B	$A + B$
Subtract	A − B	$A - B$

Operations higher up the list (e.g. power) are done before those lower down (e.g. add). This is called operator **precedence**.

Exercise: What is the result of

```
2 * 3 ** 2 - 4
```

(a) 32 (b) 10 (c) 14 (d) 2/9

Calculations

The basic operations in Python are

Power	A ** B	A^B
Multiply	A * B	$A \times B$
Divide	A / B	A/B
Add	A + B	$A + B$
Subtract	A − B	$A - B$

Operations higher up the list (e.g. power) are done before those lower down (e.g. add). This is called operator **precedence**.

Exercise: What is the result of

```
2 * 3 ** 2 - 4
```

(a) 32 (b) 10 **(c) 14** (d) 2/9

Python interprets this as (2 * (3 ** 2)) − 4

Precedence and brackets

Exercise: How do you write the following?

$$f = \frac{1}{x+1} \qquad g = \frac{1}{x} + 1$$

which one is 1/x+1 ?

Precedence and brackets

Exercise: How do you write the following?

$$f = \frac{1}{x+1} \qquad g = \frac{1}{x} + 1$$

which one is 1/x+1 ?
Divide (/) has a higher precedence than add (+), so the computer reads this as

```
g = (1/x)+1
```

Precedence and brackets

Exercise: How do you write the following?

$$f = \frac{1}{x+1} \qquad g = \frac{1}{x} + 1$$

which one is 1/x+1 ?
Divide (/) has a higher precedence than add (+), so the computer reads this as

```
g = (1/x)+1
```

If you want to change this, use brackets to tell the computer which calculations to do first

```
f = 1/(x+1)
```

Precedence and brackets

What about $h = \frac{1}{2x} + 1$? Can we write this as

```
h = 1/2*x+1
```

Precedence and brackets

What about $h = \frac{1}{2x} + 1$? Can we write this as

```
h = 1/2*x+1
```

No: The computer goes through this as before, but here the divide and multiply have the same precedence. In this case, the computer does the operations from left to right so interprets this as

```
h = (1/2)*x+1
```

i.e. $h = \frac{1}{2}x + 1$

Precedence and brackets

What about $h = \frac{1}{2x} + 1$? Can we write this as

```
h = 1/2*x+1
```

No: The computer goes through this as before, but here the divide and multiply have the same precedence. In this case, the computer does the operations from left to right so interprets this as

```
h = (1/2)*x+1
```

i.e. $h = \frac{1}{2}x + 1$
What we want is to first multiply 2 by x, so put that in brackets

```
h = 1/(2*x)+1
```

Highest precedence operations done first: ** then *,/ then +,-. If the same precedence, then goes from left to right.

$\Rightarrow$ If in doubt, put brackets around it

Scientific notation

How do we represent very large or small numbers? For example, mass of the sun (in kg)

$$m_S = 1.98892 \times 10^{30}$$

Scientific notation

How do we represent very large or small numbers? For example, mass of the sun (in kg)

$$m_S = 1.98892 \times 10^{30}$$

how about?

```
mass = 1.98892 * 10**30
```

Scientific notation

How do we represent very large or small numbers? For example, mass of the sun (in kg)

$$m_S = 1.98892 \times 10^{30}$$

how about?

```
mass = 1.98892 * 10**30
```

This will (probably) give the correct answer, but is very inefficient: it's telling the computer to do a calculation, rather than giving it a number.
Instead, programming languages use "e" notation

```
mass = 1.98892e30
```

where the "e" stands for "exponent" or "times ten to the"

Handling numbers

The way computers handle numbers is *mostly* quite straightforward, but has some quirks. Why for instance does

```
    print 7/2
```

produce the answer '3'?

Handling numbers

The way computers handle numbers is *mostly* quite straightforward, but has some quirks. Why for instance does

```
print 7/2
```

produce the answer '3'?

- This is because 7 and 2 are **integers**, whole numbers, and so the result of the calculation is also an integer.
- In computing, integers are handled very differently to numbers like 3.1415 or 2.7 which are called floating point numbers or **floats**

I can't believe you've done this! Why??

Floating point numbers

Why are integers different to floating point numbers?

- In a computer, all calculations are done in chunks of a fixed number of bits, usually 32 or 64 bits
- This means that in a single calculation a 32-bit computer can efficiently handle numbers up to 2^{32} (about 4 billion)
- This is accurate, but is also limited to whole numbers. If we want to deal with very large or non-whole numbers then we can't use integers

Floating point numbers

- The solution is to use some of those 32 bits to store the digits and the exponent separately

$$\underbrace{9.109381}_{digits} \times 10^{\underbrace{-31}_{exponent}}$$

- The downside is that we lose precision: we can only store about 6 or 7 significant places rather than 9 in an integer

Floating point numbers

- The solution is to use some of those 32 bits to store the digits and the exponent separately

$$\underbrace{9.109381}_{\textit{digits}} \times 10^{\overbrace{-31}}\;\underbrace{}_{\textit{exponent}}$$

- The downside is that we lose precision: we can only store about 6 or 7 significant places rather than 9 in an integer

- The result is that how integers and floats are represented is quite different. For example, the number 40 is

Integer	0000 0000 0000 0000 0000 0000 0010 1000
Float	0100 0010 0010 0000 0000 0000 0000 0000

- The electronic circuits which deal with floats are therefore different to those for integers

Floating point numbers

Most of the time you won't need to care about how this works: Python takes care of all this stuff for you. Some things to keep in mind though:

- If you want 7/2 to give 3.5 then make sure Python knows the numbers are floats not integers by adding a decimal point: 7./2. $\rightarrow$ 3.5
- **Exercise**: What do you think the result of this is:

```
value = 1.0e12
value = value + 1.0
```

Floating point numbers

Most of the time you won't need to care about how this works: Python takes care of all this stuff for you. Some things to keep in mind though:

- If you want 7/2 to give 3.5 then make sure Python knows the numbers are floats not integers by adding a decimal point: 7./2. $\rightarrow$ 3.5
- **Exercise**: What do you think the result of this is:

```
value = 1.0e12
value = value + 1.0
```

Should get 1 trillion and 1, but actually get just 1 trillion. Adding 1 has no effect! This is because float can't store enough digits so can't store 1.000000000001e12

Outputting results

Of course there's no point doing calculations without a way to tell anyone the result.

To output a result, Python has the **print** command

```
h = 2
print h
```

Outputting results

Of course there's no point doing calculations without a way to tell anyone the result.

To output a result, Python has the **print** command

```
h = 2
print h
```

Why "print"? Nothing is coming out the printer...

Outputting results

Of course there's no point doing calculations without a way to tell anyone the result.

To output a result, Python has the **print** command

```
h = 2
print h
```

Why "print"? Nothing is coming out the printer...

Historical reasons: now on screen rather than printer

Outputting results

When printing out results, it's good to add some text to explain what the numbers mean and what your program is doing.
In Python, text is put in quotation marks, either "like this", or 'like this'. As is traditional, printing "Hello, World!" is done using

```
print "Hello, World!"
```

You can print several things out at the same time by separating them with commas

```
print "The value h is:", h
```

Outputting results

When printing out results, it's good to add some text to explain what the numbers mean and what your program is doing.
In Python, text is put in quotation marks, either "like this", or 'like this'. As is traditional, printing "Hello, World!" is done using

```
print "Hello, World!"
```

You can print several things out at the same time by separating them with commas

```
print "The value h is:", h
```

Chunks of text in programming are called **strings**, and they can be treated like variables too

```
s = "The value h is:"
print s, h
```

These strings can be searched through, chopped up, and combined together. More on this later in the course...

Getting input

To get values from the user, Python has the **input** function (we'll come back to what functions are next time)

```
x = input("Enter x:")
h = 1./(2.*x)+1.
print h
```

This asks the user for x, calculates h and prints it out.

```
Enter x:
```

Getting input

To get values from the user, Python has the **input** function (we'll come back to what functions are next time)

```
x = input("Enter x:")
h = 1./(2.*x)+1.
print h
```

This asks the user for x, calculates h and prints it out.

```
Enter x:2
1.25
```

Case sensitive revisited

As with variable names, Python commands are **case sensitive**. Therefore, although

```
Print h
```

looks fine to a human, but to Python it's incomprehensible.

Case sensitive revisited

As with variable names, Python commands are **case sensitive**. Therefore, although

```
Print h
```

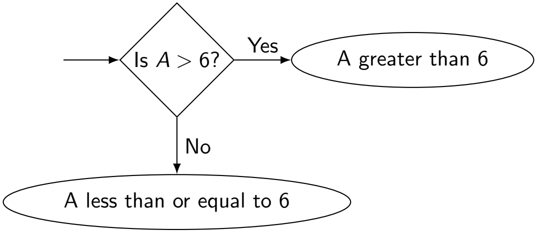
looks fine to a human, but to Python it's incomprehensible. Humans are very good at seeing patterns and filling in gaps. Computers are still very bad at these things so need precise input.

In programming you have to be very picky about small details. Even a little misplaced comma or capital letter can break your program

When something like this happens, the computer will try to help and tell you where it thinks the mistake is. Often some detective work is needed.

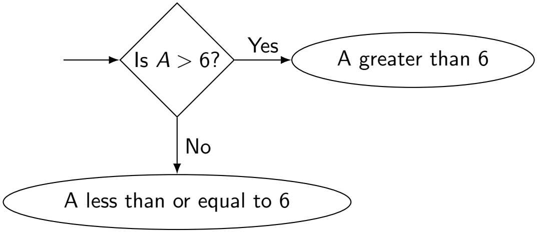
Making decisions

One of the important things we want an algorithm to do is make decisions



Making decisions

One of the important things we want an algorithm to do is make decisions



In Python, this is done using **if** clauses

```
if A > 6:
    print "A is greater than 6"
else:
    print "A is less than or equal to 6"
```

Making decisions

The format is

```
if condition:
    commands
else:
    commands
```

where condition is what you want to test, and commands is one or more lines of commands, called a code **block**.
For example

```
if A > 6:
    B = 2
    print "Got spam"
else:
    B = -2
    print "Got eggs"
```

Making decisions

The format is

```
if condition:
    commands
else:
    commands
```

where condition is what you want to test, and commands is one or more lines of commands, called a code **block**.
Most important: Note that the code blocks are indented

Indentation is crucial in Python, and is how it knows where blocks start and end

Making decisions

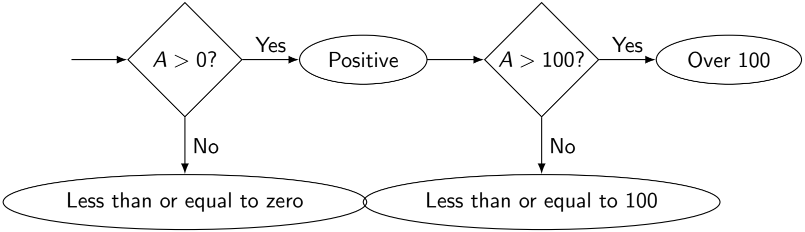
Exercise: Draw the flow diagram for the following Python code

```
if A > 0:
    print "positive"
    if A > 100:
        print "Over 100"
    else:
        print "Less than or equal to 100"
else:
    print "Less than or equal to zero"
```

Making decisions

Exercise: Draw the flow diagram for the following Python code

```
if A > 0:
    print "positive"
    if A > 100:
        print "Over 100"
    else:
        print "Less than or equal to 100"
else:
    print "Less than or equal to zero"
```



More complicated decisions

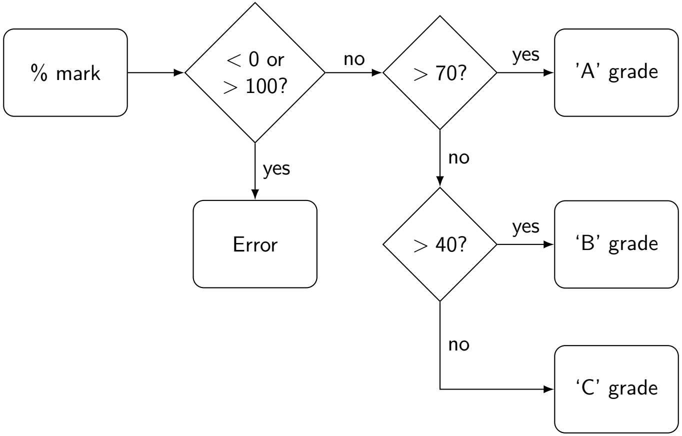
So far we've only used > to test if one number is greater than another. You can use the following comparisons

A greater than B	A > B
A less than B	A < B
A greater than or equal to B	A >= B
A less than or equal to B	A <= B
A equal to B	A == B

A = B sets A equal to B, but A == B tests if A is equal to B

More complicated decisions

What about our original percentage to grade problem?



More complicated decisions

In Python, this can be written as

```
m = input("Enter percentage mark: ")
if m < 0 or m > 100:
    print "Error"
    exit()

if m > 70:
    print "A"
else:
    if m > 40:
        print "B"
    else:
        print "C"
```

Note that **if** doesn't need an **else** - you can just miss it out
Also you can combine tests using **and**, **or** and **not**

More complicated decisions

A shorthand way to check for several possibilities is to use **elif** instead of **else** and **if**

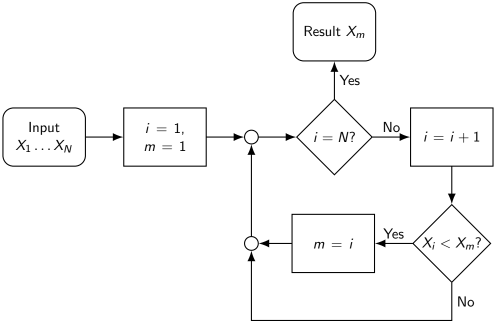
```
m = input("Enter percentage mark: ")
if m < 0 or m > 100:
    print "Error"
    exit()

if m > 70:
    print "A"
elif m > 40:
    print "B"
else:
    print "C"
```

Going around in circles

Another common task we want is to be able to repeat commands while some condition is true.

Going around in circles

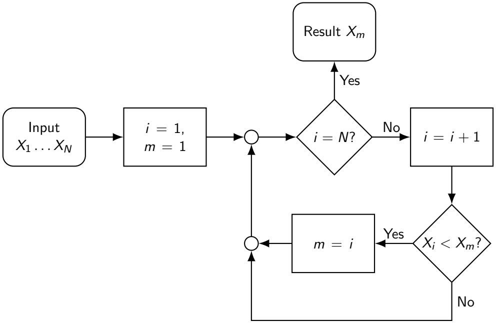
Another common task we want is to be able to repeat commands while some condition is true.
From last time: find the lowest number *m* in a list of *N* numbers



Going around in circles

Another common task we want is to be able to repeat commands while some condition is true.

From last time: find the lowest number *m* in a list of *N* numbers



```
i = 0
m = 0
while i < N:
    i = i + 1
    if X[i] < X[m]:
        m = i
print "Min is ", X[m]
```

Going around in circles

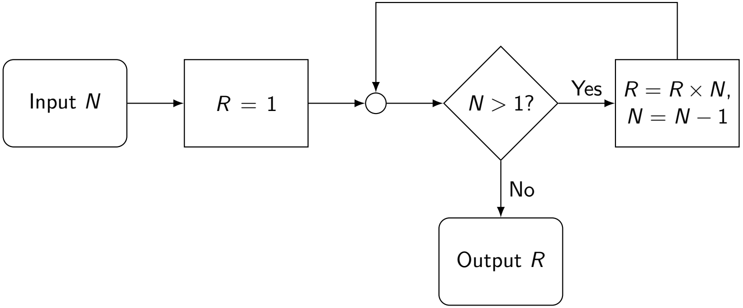
A few new things in this example

```
i = 0
m = 0
while i < N:
    i = i + 1
    if X[i] < X[m]:
        m = i
print "Min is ", X[m]
```

- **while** repeats the commands in its block (indented) as long as its condition is true. In this case, keeps going until i is equal to N
- In programming, lists of numbers $X_1 \ldots X_N$ called **arrays** can be used. X_1 is written X[1] so X_m is X[m]
- In Python, arrays start from 0, so in an array of N numbers, the first one is X[0] and the last is X[N−1]

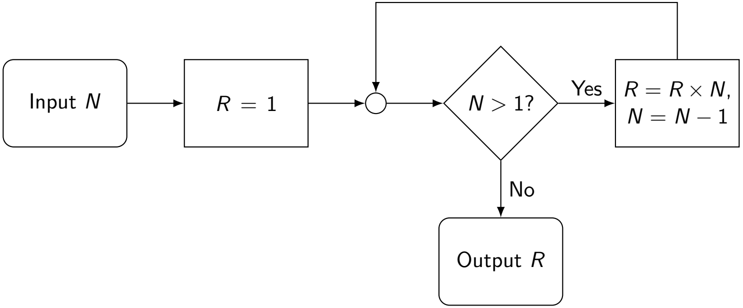
Going around in circles

Exercise: Try writing down the Python code for this loop which calculates the factorial $N!$ of a number N



Going around in circles

Exercise: Try writing down the Python code for this loop which calculates the factorial $N!$ of a number N



```
N = input("Enter N: ")
R = 1
while N > 1:
    R = R * N
    N = N - 1
print "Result is ", R
```

More advanced topics

- We have covered the most basic parts of programming, and almost everything can be written with just **if** and **while** blocks
- For bigger programs though this would become very unwieldy, slow to write and difficult to understand

More advanced topics

- We have covered the most basic parts of programming, and almost everything can be written with just **if** and **while** blocks
- For bigger programs though this would become very unwieldy, slow to write and difficult to understand

- There are many more abstract ideas in programming, most of which are ways to handle increasingly complicated programs
- In this course we won't be able to cover all of them, but you will come across some of the more common ones
- **Next time (lab 1)**: more loops, lists, arrays, and functions

Summary

- Python is an efficient way to express algorithms, easy to read but also a “proper” language

Summary

- Python is an efficient way to express algorithms, easy to read but also a “proper” language
- Programming languages have some quirks due to how computers work

Summary

- Python is an efficient way to express algorithms, easy to read but also a “proper” language
- Programming languages have some quirks due to how computers work
- Computers are very picky about small details such as punctuation (commas, colons etc.) and upper vs. lower case

Summary

- Python is an efficient way to express algorithms, easy to read but also a "proper" language
- Programming languages have some quirks due to how computers work
- Computers are very picky about small details such as punctuation (commas, colons etc.) and upper vs. lower case
- Python is particularly picky about indentation: changes in indentation mark the beginning and end of blocks of code

Summary

- Python is an efficient way to express algorithms, easy to read but also a “proper” language
- Programming languages have some quirks due to how computers work
- Computers are very picky about small details such as punctuation (commas, colons etc.) and upper vs. lower case
- Python is particularly picky about indentation: changes in indentation mark the beginning and end of blocks of code
- See the blue boxes in the slides for the most important things to watch out for

Enough!

http://www-users.york.ac.uk/∼bd512/teaching.shtml

- Python is an efficient way to express algorithms, easy to read but also a "proper" language
- Programming languages have some quirks due to how computers work
- Computers are very picky about small details such as punctuation (commas, colons etc.) and upper vs. lower case
- Python is particularly picky about indentation: changes in indentation mark the beginning and end of blocks of code
- See the blue boxes in the slides for the most important things to watch out for