Statements 2

- Note: `i=i+1` is a valid statement. Don't confuse it with an equation `i==i+1` which is always false for normal numbers.
- The statement `i=i+1` is a very common *idiom*: it just increments by one the value pointed by variable `i`.
- The shorthand for this is: `i += 1`. Similarly, `i=i-1` can be written with `i-=1`
- The binary operators `+ - * / // % & | ^ >> << **` have also corresponding *augmented assignment operators* `+= -= *= /= //= %= &= |= ^= >>= <<= **=`
- The following forms are equal
  - `a` *operator*`= b`
  - `a = a` *operator* `b`

Doing repetitive things: *loops* 1

- In Python there are two kinds of loops: `while` and `for`
- A `while` loop repeats a set of statements while a given condition holds
- Example:

```
i=1
while i*i < 1000:
    print "Square of", i, "is", i*i
    i=i+1
print "Finished printing all the squares below 1000"
```

Doing repetitive things: *loops* 2

- This `while` loop repeated two statements: the `print` statement and the assignment statement, which increments the value of variable `i` by one.
- The block of staments that is subjected to a compound statement, like the while statement, are denoted by *indentation*
- Indentation means that a line is started with a sequence of whitespaces, usually this is a sequence of four space characters
- All statements in the same block must start at the same column

Doing repetitive things: *loops* 3

- The last `print` statement has less indentation than the previous two lines, which means the block of statements has ended
- Another way to repeat statements is with the `for` statement
- Example:

```
sum=0
for i in [0,1,2,3,4,5,6,7,8,9]:
    sum=sum+i
print "The sum is", sum
```

Doing repetitive things: *loops* 4

- The `for` loop executes the statements in the block as many times as there elements in the sequence given after the keyword `in`
- And at each iteration, the variable `i` has another value from the sequence
- The elements of the sequence are guaranteed to be iterated in order
- In the previous example the sequence happened to be a `list`, which is a basic data structure of Python. The list is one example of sequences
- Even more generally, instead of a sequence the elements can come from an *iterable*. They are a generalisation of a sequence.

The break statement

- Sometimes, when we are executing a loop, we want to stop the iteration, even when all elements have not been iterated or when the `while` condition still holds
- This can be achieved by executing the `break` statement in the loop body
- This stops the loop and continues execute whatever comes after the loop

Continue

- `continue` is another loop control statement
- This statement stops the execution of statements of the loop body that follow the continue statement
- The execution then continues from the beginning of the loop body, and if the loop is a for loop, the next value in the sequence will be processed

break example

```
for i in [1,2]:
    print i
    print "hello"
    break
    print "bye"
print "The for loop just finished"
```

continue example

```
for i in [1,2]:
    print i
    print "hello"
    continue
    print "bye"
print "The for loop just finished"
```

Making decisions: the `if` statement 1

- Let's say we want to find the absolute value of a number x given by user
- The absolute value is x itself if it is non-negative, or if x is negative then the absolute value is its opposite: -x

Making decisions: the if statement 1

- Let's say we want to find the absolute value of a number x given by user
- The absolute value is x itself if it is non-negative, or if x is negative then the absolute value is its opposite: -x
- We can express this decision with the if statement:

```
import sys
str=sys.stdin.readline()
x=int(str)
print "The absolute value of", x, "is"
if x >= 0:
    print x
else:
    print -x
```

Making decisions: the `if` statement 2

- The `if` statement is a *compound statement* like `for` and `while` statements
- But the if statement can have several blocks of statements subjected to it:
- In the previous example there was one block in the `if`-clause, and one block in the `else`-clause

Making decisions: the if statement 3

- Generic form of an if statement is:

```
if condition1:
    statement1_1
    statement1_2
    ...
elif condition2:
    statement2_1
    statement2_2
    ...
...
else:
    statementn_1
    statementn_2
    ...
```

Making decisions: the if statement 4

- Example:

```
c=-5
if c > 0:
    print "c is positive"
elif c<0:
    print "c is negative"
else:
    print "c is zero"
```

- Python doesn't have a switch-case statement, but the `if/elif/else` statement can be used to imitate switch-case

An else clause at the end of a loop

- The block under else is executed always when the loop wasn't stopped by the `break` statement
- Example:

```
for c in "alongstring":
    if c == 'q':
        break
else:
    print "Character q not found"
```

Functions 1

- The loop structures of Python allow block of statements to be reused
- So, instead of writing

```
print "Hoorray!"
print "Hoorray!"
print "Hoorray!"
```

- we can write

```
for i in range(3):
    print "Hoorray!"
```

- But it would be nice to be able to use a block of statements also elsewhere in the source code, not just during the execution of the loop

Functions 2

- A function is a block of statements, and we can refer to it by a name
- A function also has an interface, so we can give parameters to it and receive a value from it.
- Because the behaviour of a function can depend on the parameters given to it, the resulting function is even more reusable.
- Python functions can be like mathematical functions that only compute a new value based on given rules and the parameters
- Or the functions can in addition have side-effects

Functions 3: function definition

- A function definition looks like this:

```
def function_name(param1, ..., paramn,
                  dparam1=expr1, ...,dparamm=exprm):
    "Documentation of the function goes here"
    statement1
    ...
    statementk
    return expression
```

Functions 4

- The parameter names can be used in the statements of the block
- With the `return` statement a function can return a value back to the caller of the function. The return statement is not mandatory since a function doesn't need to return a value.
- The line that defines the function name and its parameters is called the *signature* of the function.
- After the signature there is an optional *documentation string*. This exists to give information about the function to a user. This information can be retrieved with the call `help(function_name)`

Functions 5: calling a function

- When Python sees a function definition, it just stores the function. It doesn't execute the statements yet.
- The execution of a function is triggered by a *function call*
- When Python sees a function call `function_name(expr1, ..., exprl)` the parameters of the function are bound to the values of the expressions given in the function call. And then the statements are executed
- After this the value of the *function call expression* is bound to the value returned by the function.

Functions 6: optional parameters

- In the function definition some of the parameters were given a *default value*: dparam=expression
- These are *optional parameters*, they can only appear after the mandatory parameters
- If in a function call some of the optional parameters from the end are left out, they are given the default parameter values
- For example, if the signature is `f(a, b=1, c=2)`, and the call is *f(0,2)*, then the parameter values are a=0, b=2, c=2.

Functions 7: named arguments

- Normally the function call arguments correspond to function parameters in the same order (*positional arguments*).
- The order can be changed using *named arguments* in the function call.
- For example, the function call `f(0,c=3)` will result in the parameter bindings a=0,b=1,c=3.
- The named arguments in the function call must come after any positional arguments

Sequences

- A *list* is a dynamic collection of values/objects that are stored in order
- An example of a list with 4 values: [2, 100, "hello", 1.0]
- A tuple is fixed length, immutable, ordered container. Elements of tuple are separated by commas and written between parentheses
- A singleton (3,), a pair (1,3), and a triple (1,"hello",1.0) are examples of tuples
- As we can see, both lists and tuples can contain values of different type
- Lists, tuples and strings are called *sequences* in Python, and they have several commonalities

Common features of sequences

- The length of any sequence can be found with the `len` function
- `min` and `max` functions find the minimum and maximum element of a sequence, and `sum` adds all the elements of a sequence of numbers together
- Sequences can be concatenated with the + operator, and repeated with the * operator: `"hi"*3=="hihihi"`
- Since sequences are ordered, we can refer to the elements of a sequence by integers using the bracket notation: `"abcd"[2]=="c"`
- Note that the indexing begins from 0
- Negative integers start indexing from the end: -1 refers to the last element, -2 refers to the second last, and so on