The lecture starts by discussing different ways to write a program for computing

$$\sum_{i=0}^{n} i$$
Definition
A *conditional expression* is an expression that may have the *Boolean* value *True* or *False*.
Definition
A *conditional expression* is an expression that may have the *Boolean* value *True* or *False*.

Some common operators that yield conditional expressions are:

- `==`, `!=`
- `<`, `>`, `<=`, `>=`
- One combines different Boolean values with `or` and `and`
- `not` gives the *logical negation* of the expression that follows
Conditional Expression Examples

Example

```python
2 >= 4  # False
2 < 3 < 4 # True
2 < 3 and 3 < 2 # False
2 != 3 < 4 or False # True
2 <= 2 and 2 >= 2 # True
not 2 == 3 # True
not False or True and False # True!
```

Note in the last example the priority rules when using `not`, `and`, and `or`.
They can be compared to addition (→ `or`), multiplication, (→ `and`), and sign change (→ `not`) in classical arithmetics.
Conditional statements

*If statement*

A conditional statement delimits a *block* that will be executed if the condition is true. An *optional* block, started with the keyword *else* will be executed if the condition is not fulfilled.

*Example*

We print the absolute value of \( x \):

Mathematics:

\[
|x| = \begin{cases} 
  x & \text{if } x \geq 0 \\
  -x & \text{else}
\end{cases}
\]

Python:

```python
x = ...
if x >= 0:
    print(x)
else:
    print(-x)
```
Error messages

By using Boolean expressions you can check for potential errors:

denom = ...  
num = 3  
if not denom == 0:  
    frac = num / denom  
else:  
    raise ZeroDivisionError("Don’t divide by zero!")

The string in the parenthesis is up to the programmer.  
The error type has to be predefined.

See later chapter.
The full **for** statement: **break**

**break** gets out of the for loop even if the list we are iterating is not exhausted.

```python
for x in x_values:
    if x > threshold:
        break
    print(x)
```
The full for statement: `else`

`else` checks whether the for loop was `broken` with the `break` keyword.

```python
for x in x_values:
    if x > threshold:
        break
else:
    print("all the x are below the threshold")
```

If we `did not break` the `else`-block is executed.
Basic string formatting

- Using keywords:

```
"I’m a {something}.".format(something="string")
# "I’m a string."
"I’m a {something}.".format(something=10)
# "I’m a 10."
```

- Using positional arguments:

```
"Two strings, {} and {}.".format(’a’, ’b’)
# ’Two strings, a and b.’
"Pi = {0} and e = {1}.".format(pi, e)
# ’Pi = 3.14159265359 and e = 2.71828182846.’
```

- Numbers can be formatted:

```
quantity = 33.45
"{0:f}".format(quantity)  # 33.450000
"{0:1.1f}".format(quantity)  # 33.5
"{0:.2e}".format(quantity)  # 3.35e+01
```
Old string formatting

Older versions of Python use the following syntax to format strings. It works in newer versions too.

- **for strings:**
  ```python
course_code = "NUMA21"
print("This course’s name is %s" % course_code)
# This course’s name is NUMA21
```

- **for integers:**
  ```python
nb_students = 16
print("There are %d students" % nb_students)
# There are 16 students
```

- **for reals:**
  ```python
average_grade = 3.4
print("Average grade: %f" % average_grade)
# Average grade: 3.400000
```
Consider the following mathematical *function*:

\[ x \mapsto f(x) := 2x + 1 \]

The Python equivalent is:

```python
def f(x):
    return 2*x + 1
```

- the keyword `def` tells Python we are defining a function
- `f` is the *name* of the function
- `x` is the *parameter*, or *input* of the function
- what follows `return` is called the *output* of the function
Calling a Function

Once the following function is defined:

```python
def f(x):
    return 2*x + 1
```

it may now be called using:

```python
f(2) # 5
f(1) # 3
# etc.
```

Note, the *parameter* $x$ is replaced by the *argument* 2.
Collect a sequence commands in a file with its name ending in `.py`, e.g. `smartscript.py`

```python
def f(x):
    return 2*x + 1

z = []
for x in range(10):
    if f(x) > pi:
        z.append(x)
    else:
        z.append(-1)
print(z)
```

From IPython:
```
run smartscript.py  # [-1, -1, 2, 3, 4, 5, 6, 7, 8, 9]
```

From command prompt (i.e. in the terminal window of your computer)
```
python smartscript.py
```
Create a *module* by collecting functions into a single file, e.g. `smartfunctions.py` as:

```python
def g(x):
    return x**2 + 4 * x - 5
def h(x):
    return 1/f(x)
def f(x):
    return 2 * x + 1
```

▶ These functions can now be used by any external script or directly in the IPython environment.
▶ Functions within the module can depend on each other.
▶ Grouping functions with a common theme or purpose gives modules that can be shared and used by others.
Using modules

Modules need to be imported.

```python
import smartfunctions
print(smartfunctions.f(2))  # 5

from smartfunctions import g  # import just only g
print(g(1))  # 0

from smartfunctions import *  # import all
print(h(2) * f(2))  # 1.0
```

Note the use of the namespace smartfunctions!
You have already used the command `plot`. It needs a list of $x$ values and a list of $y$ values. If a single list, $y$ is given, the list `list(range(len(y)))` is assumed as $x$ values.

You may use the keyword argument `label` to give your curves a name, and then show them using `legend`.

```python
x_vals = [.2*n for n in range(20)]
y1 = [sin(.3*x) for x in x_vals]
y2 = [sin(2*x) for x in x_vals]
plot(x_vals, y1, label='0.3')
plot(x_vals, y2, label='2')
legend()
show()  # not always needed.
```
Here comes the plot with a legend.

Here is an example with more keywords:

```python
plot(x_vals, y2,
     color='green',
     linestyle='dashed',
     marker='o',
     markerfacecolor='blue',
     markersize=12, linewidth=6)
```
Plot (Cont.)

... and here the resulting graph: