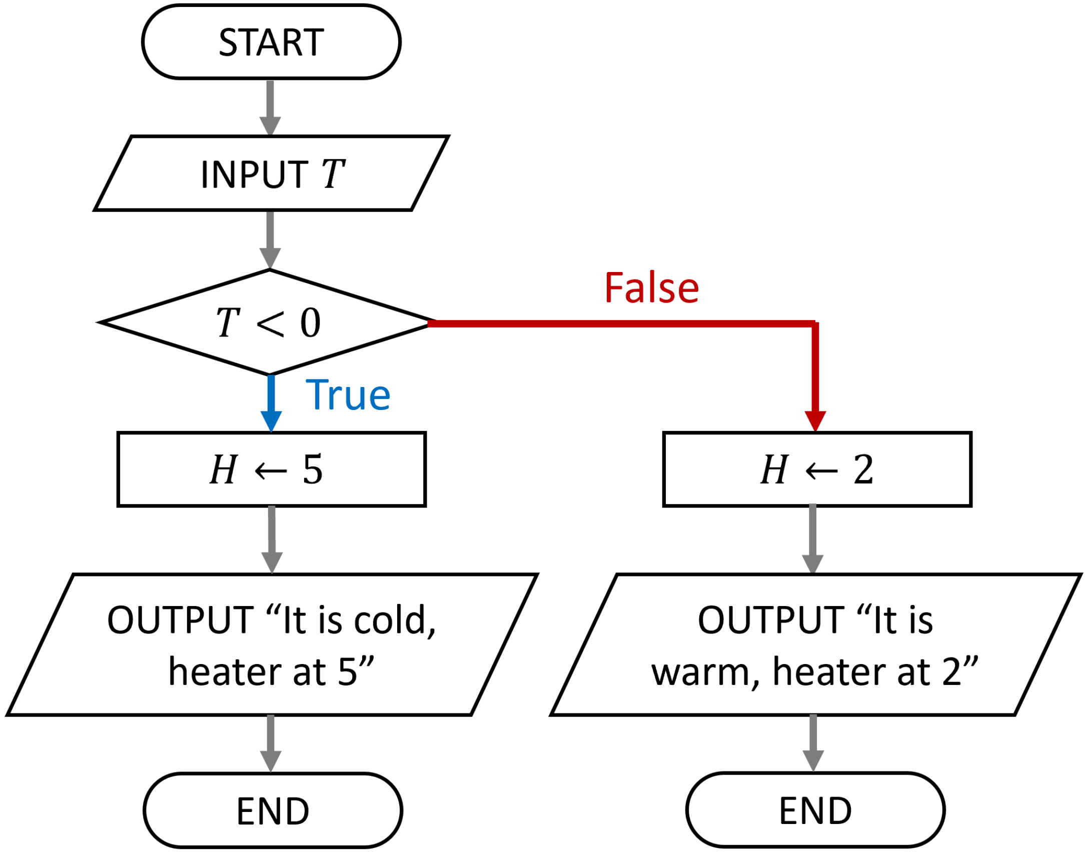
PHOT 110: Introduction to programming

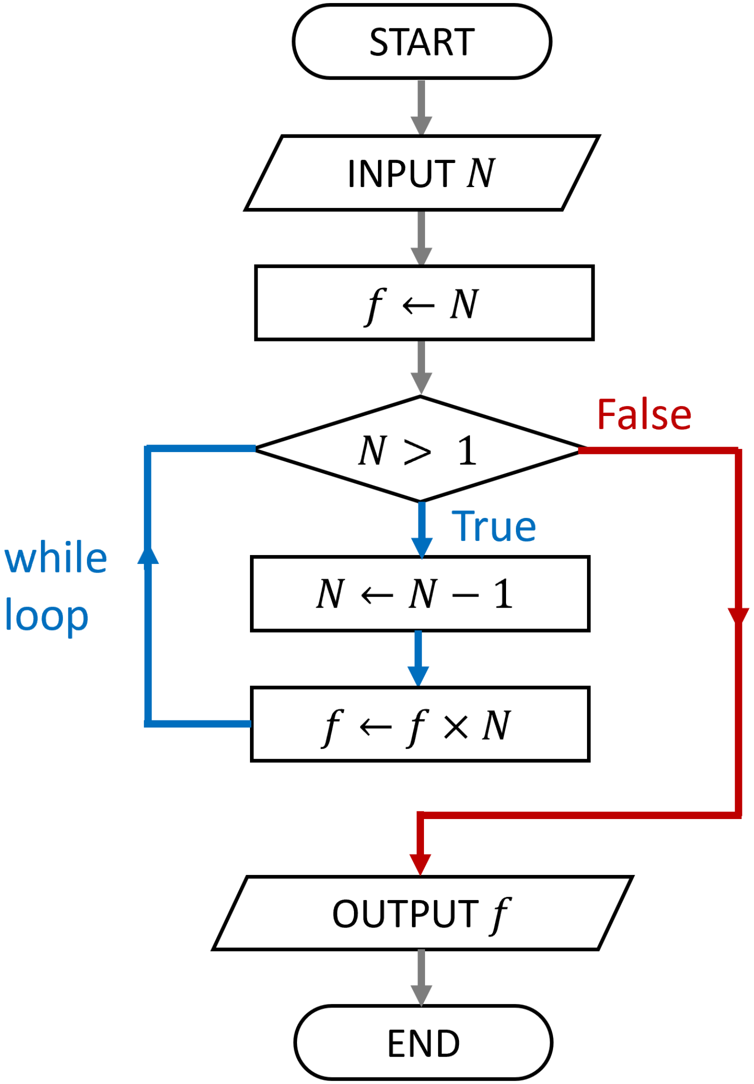
LECTURE 03

Michaël Barbier, Spring semester (2023-2024)

CONTROL FLOW: CONDITIONAL BRANCHING AND LOOPS

- Branching & loops





CONTROL FLOW: CONDITIONAL BRANCHING AND LOOPS

- Branching if/else statements
- The while loop
- Lists of objects
- The for loop: iterating over a list

CONDITIONAL BRANCHING: IF/ELIF/ELSE

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
  - An **`if`** code-block is executed when the condition is `True`
  - An **`else`** code-block is executed when the condition is `False`

```
if <condition>:
    <statement>
    ...
```

```
if <condition>:
    <statement>
    ...
else <condition>:
    <statement>
    ...
```

CONDITIONAL BRANCHING

- **`elif`** keyword acts as a **`else if`**
- multiple **`elif`** statements can follow an **`if`**, with an optional **`else`**

```
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
else <condition>:
    <statement>
    ...
```

```
if <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
elif <condition>:
    <statement>
    ...
```

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
  - An **`if`** code-block is executed when the condition is `True`

```python
age = 46
if age >= 16:
  print("You can drive a tractor")   # if code-block
```

```
You can drive a tractor
```

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
    - **`if`** code-block is executed when the condition is **`True`**
- The **indented code-block** can contain multiple statements

```python
speed_limit = 120
speed = 137
if speed > speed_limit:
  speed_diff = speed - speed_limit
  print(f"You drive {speed_diff} km/h too fast")
```

```
You drive 17 km/h too fast
```

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
  - **`if`** code-block is executed when the condition is **`True`**
  - **`else`** code-block is executed when the condition is **`False`**

```python
age = 11
if age > 18:
  print("You can drive a car")
else:
  print("You should take the bus")
```

```
You should take the bus
```

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
  - **`if`** code-block is executed when the condition is **`True`**
  - **`else`** code-block is executed when the condition is **`False`**
  - **`elif`** keyword acts as a **`else if`**

```python
age = 17
if age > 18:
  print("You can drive a car")
elif age > 16:
  print("You can drive a tractor")
else:
  print("You can ride a bicycle")
```

```
You can drive a tractor
```

CONDITIONAL BRANCHING

- Run **code-blocks** according to a **condition**
  - **`if`** code-block is executed when the condition is `True`
- The **indented code-block** can contain multiple statements
- indentation is the same within a code-block

```python
1 age = 19
2 if age > 18:
3     print("You can drive a car")    # This line is indent
4   print("You can drive a bicycle")
5   print("You can drive a tractor")
```

```
IndentationError: unindent does not match any outer
indentation level (<string>, line 4)
```

WHILE LOOP

THE WHILE LOOP

- Repeats code-block until the **condition** is **`False`**
- A **`while` loop** is used when:
  - we don’t know how many iterations we need, and
  - we have a stopping criterium/condition

```
while <condition>:
  <statement>
  <statement>
  <statement>
  ...
  <statement>
```

THE WHILE LOOP

- Repeats code-block until the **condition** is **`False`**
- A **`while` loop** is used when:
  - we don’t know how many iterations we need, and
  - we have a stopping criterium/condition

```python
t = 0; t_max = 10
while t < t_max:
  t = t + 3.86
  print(f"The elapsed time is: {t:5.3} s")
print("End of the program")
```

```
The elapsed time is:  3.86 s
The elapsed time is:  7.72 s
The elapsed time is:  11.6 s
End of the program
```

THE WHILE LOOP

- Repeats code-block until the **condition** is **`False`**
- Can get in an infinite loop !
  - Stop the program with the stop button, in a terminal press key combination **`Ctrl`** + **`c`**
  - Adapt the stopping criterium/condition

```python
n = 0
while n > -100:
n = n + 1
print(f"The current number is: {n}")
```

PYTHON LISTS

LISTS OF OBJECTS

- A list can contain several objects
- The object types can be different
- Lists are also objects

```python
# A list with mixed object types
my_list_of_objects = ["It's Monday", False, 34, 23.4]

# Lists can be elements of a list
a_list_with_a_list = [5, 10.5, ["green", "red"], True]
print(a_list_with_a_list)
```

```
[5, 10.5, ['green', 'red'], True]
```

LISTS OF OBJECTS

- Length of the list is the number of elements applying the **`len()`** function: `len(a_list)`
- The object types can be different
- Lists are also objects

```python
# A list with mixed object types
my_list_of_objects = ["It's Monday", False, 34, 23.4]

# Lists can be elements of a list
a_list_with_a_list = [5, 10.5, ["green", "red"], True]
print(a_list_with_a_list)
```

```
[5, 10.5, ['green', 'red'], True]
```

APPENDING AN ELEMENT TO A LIST

- Append an element to the end of a list
- Length (number of elements) of the list increases with one

```python
# Printing the first and then the second element
a_list = ["First", False, 34, 23.4]
print(a_list)
print(f"The length of the list = {len(a_list)}")
a_list.append("extra_element")
print(a_list)
print(f"The length of the adapted list = {len(a_list)}")
```

```
['First', False, 34, 23.4]
The length of the list = 4
['First', False, 34, 23.4, 'extra_element']
The length of the adapted list = 5
```

MORE METHODS OF LIST

- We use the dot-notation: `the_list.append(the_element)`
- This notation is to call a method on an object
- We will see how to make our own methods (and classes) later in the chapter on object oriented programming
- There are more methods we can make use of, see https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

```
a_list = ["First", False, 34, 5, 34]   # Define the list
a_list.remove(34)                      # Remove first 34
a_list.insert(3, "inserted_string")    # Insert str
print(a_list)                          # Print the list
```

```
['First', False, 5, 'inserted_string', 34]
```

SELECTING ELEMENTS IN A LIST

- Select an element of a list by its index
- syntax for indexing: `a_list[element_index]`
- index is zero-based
- negative index starts from the end of the list

```python
# Printing the first and then the second element
a_list = ["First", False, 34, 23.4]
print(a_list[0])
print(a_list[1])
```

```
First
False
```

SELECTING ELEMENTS IN A LIST

- Select an element of a list by its index
- syntax for indexing: `a_list[element_index]`
- index is zero-based
- negative index starts from the end of the list

```python
# Using negative indexing
a_list = ["First", False, 34, 23.4]
print(a_list[-1])
```

```
23.4
```

SLICING A LIST

- Selecting multiple elements is called **slicing**
- syntax for slicing: `a_list[start:stop_exclusive]`

```python
# A list with mixed object types
a_list = [23, 45, 65, 78, 92, 100, 102, 105 ]
print(a_list[2:5])
```

```
[65, 78, 92]
```

SLICING A LIST

- Selecting multiple elements is called **slicing**
- syntax for slicing: `a_list[start:stop_exclusive]`

```python
# An empty start_index starts from the first index
a_list = [23, 45, 65, 78, 92, 100, 102, 105 ]
print(a_list[:5])
```

```
[23, 45, 65, 78, 92]
```

```python
# An empty end_index end at the last index
a_list = [23, 45, 65, 78, 92, 100, 102, 105 ]
print(a_list[3:])
```

```
[78, 92, 100, 102, 105]
```

SLICING A LIST

- Selecting multiple elements is called **slicing**
- syntax for slicing: `a_list[start:stop_exclusive]`
- Additional step parameter: `a_list[start:stop_exclusive:step]`

```python
# Take every second element by stepping
a_list = [23, 45, 65, 78, 92, 100, 102, 105 ]
print(a_list[1::2])
```

```
[45, 78, 100, 105]
```

THE FOR LOOP

THE FOR LOOP

- To iterate: to repeat a process
- A **for** loop can be used when:
  - the number of **iterations** is known, or
  - we iterate over a list of elements

```
for <element> in <list>:
    <statement>
    <statement>
    ...
    <statement>
```

THE FOR LOOP

- To iterate: to repeat a process
- A **for** loop can be used when:
  - the number of **iterations** is known, or
  - we iterate over a list of elements

```python
# Print all elements of a list
days = ["Mon", "Tue", "Wed", "Thu", "Fri"]
for el in days:
  print(el)
```

```
Mon
Tue
Wed
Thu
Fri
```

INTERMEZZO: USING RANGE()

- A **range** is a sequence type (like `list`) for integer numbers
- Construct it using: `range(start, stop_exclusive, step)`
- It is convenient for **for** loop
- See also: https://docs.python.org/3/library/stdtypes.html#range

```python
# A list with mixed object types
a_range = range(1, 10, 2)  # Construct a range
print(a_range)             # Lazy evaluated
print(list(a_range))       # Converted to a list
```

```
range(1, 10, 2)
[1, 3, 5, 7, 9]
```

THE FOR LOOP

- To iterate: to repeat a process
- A **for** loop can be used when:
  - the number of **iterations** is known, or
  - we iterate over a list of elements

```python
# Use the range function to get a sequence of numbers
for i in range(1,10,2):
  print(i)
```

```
1
3
5
7
9
```

THE FOR LOOP

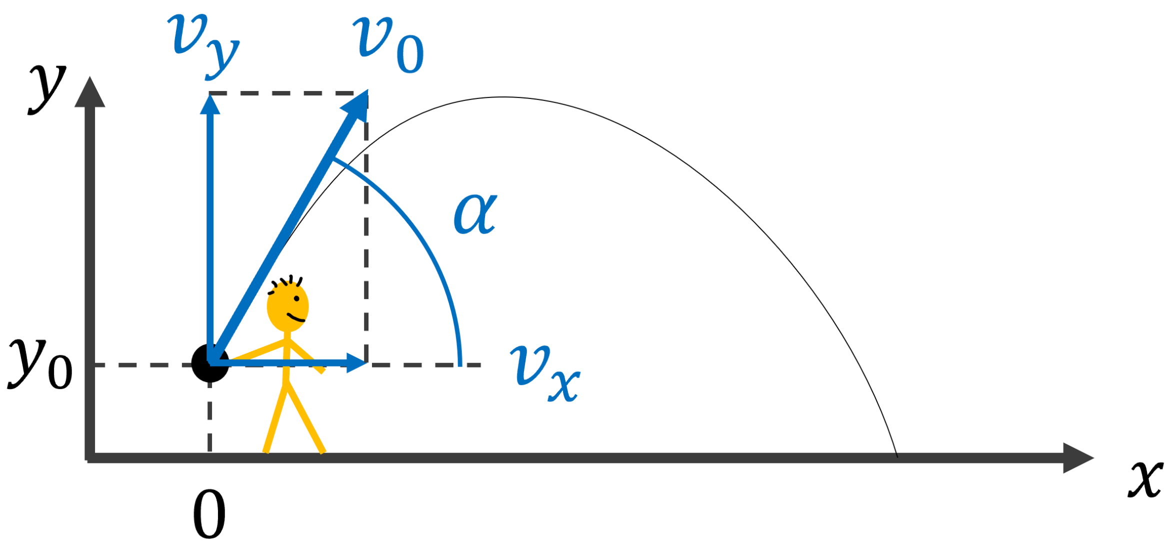
- To iterate: to repeat a process
- A **for** loop can be used when:
    - the number of **iterations** is known, or
    - we iterate over a list of elements

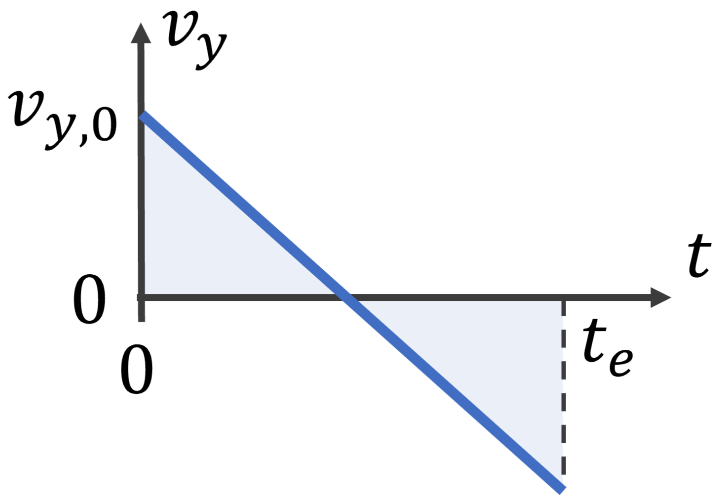
```python
# Use the range function to get indices
days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
for i in range(0,len(days),2):
  print(days[i])
```

```
Mon
Wed
Fri
Sun
```

EXAMPLE SCIENTIFIC ALGORITHM

NUMERICAL APPROX. OF THE TRAJECTORY OF A BALL





- Gravitation: $g = 9.81\ \mathrm{m/s^2}$
- Air resistance: ignore for a slow heavy ball
- horizontal velocity is constant

TRAJECTORY OF A BALL

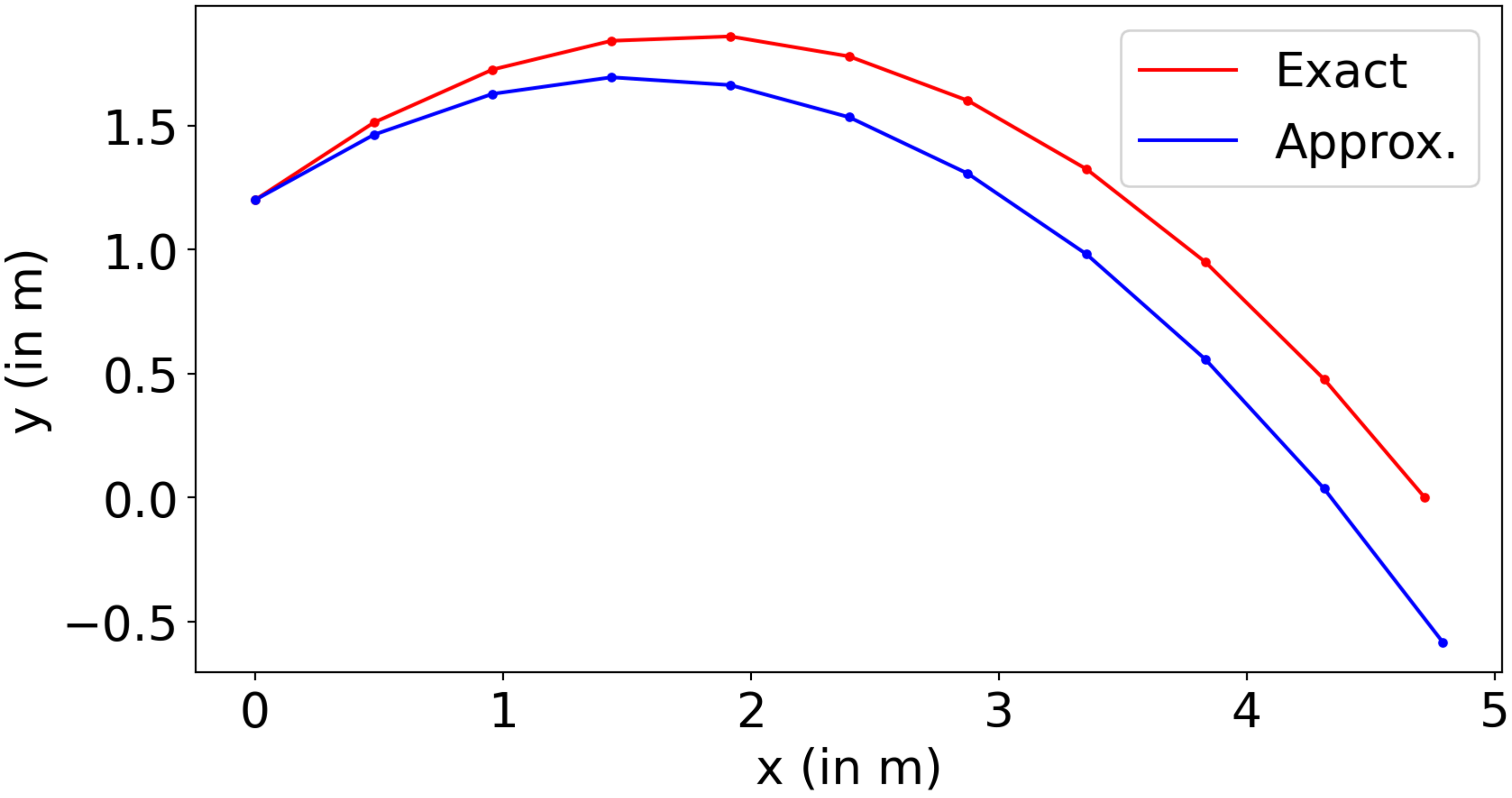
```python
# Load library for sine and cosine
import math

# Algorithm parameters in MKS units
v0 = 6
angle_in_degrees = 37
g = 9.81
x = 0; y = 1.20; t = 0

# Calculate initial velocity in x
#     and y directions
angle = angle_in_degrees * math.pi/180
vx = v0 * math.cos(angle)
vy = v0 * math.sin(angle)
```

TRAJECTORY OF A BALL

```
t = 0.10: (0.48,1.46)
t = 0.20: (0.96,1.63)
t = 0.30: (1.44,1.69)
t = 0.40: (1.92,1.66)
t = 0.50: (2.40,1.53)
t = 0.60: (2.88,1.31)
t = 0.70: (3.35,0.98)
t = 0.80: (3.83,0.56)
t = 0.90: (4.31,0.04)
t = 1.00: (4.79,-0.58)
```

NUMERICAL APPROXIMATION ERRORS

Comparison with exact trajectory obtained before



SUMMARY

- Control flow exists of
  - Branching if/else
  - `while`/`for` loops
- Allows implementing complex algorithms
- Lists are versatile data structures
- The `for` loop: iterating over list elements
- More complex **scientific algorithms**:
  - Iterative methods
  - Stopping criteria