Introduction to Python

Lab 1

Kwabena Owusu, Peter Steiglechner

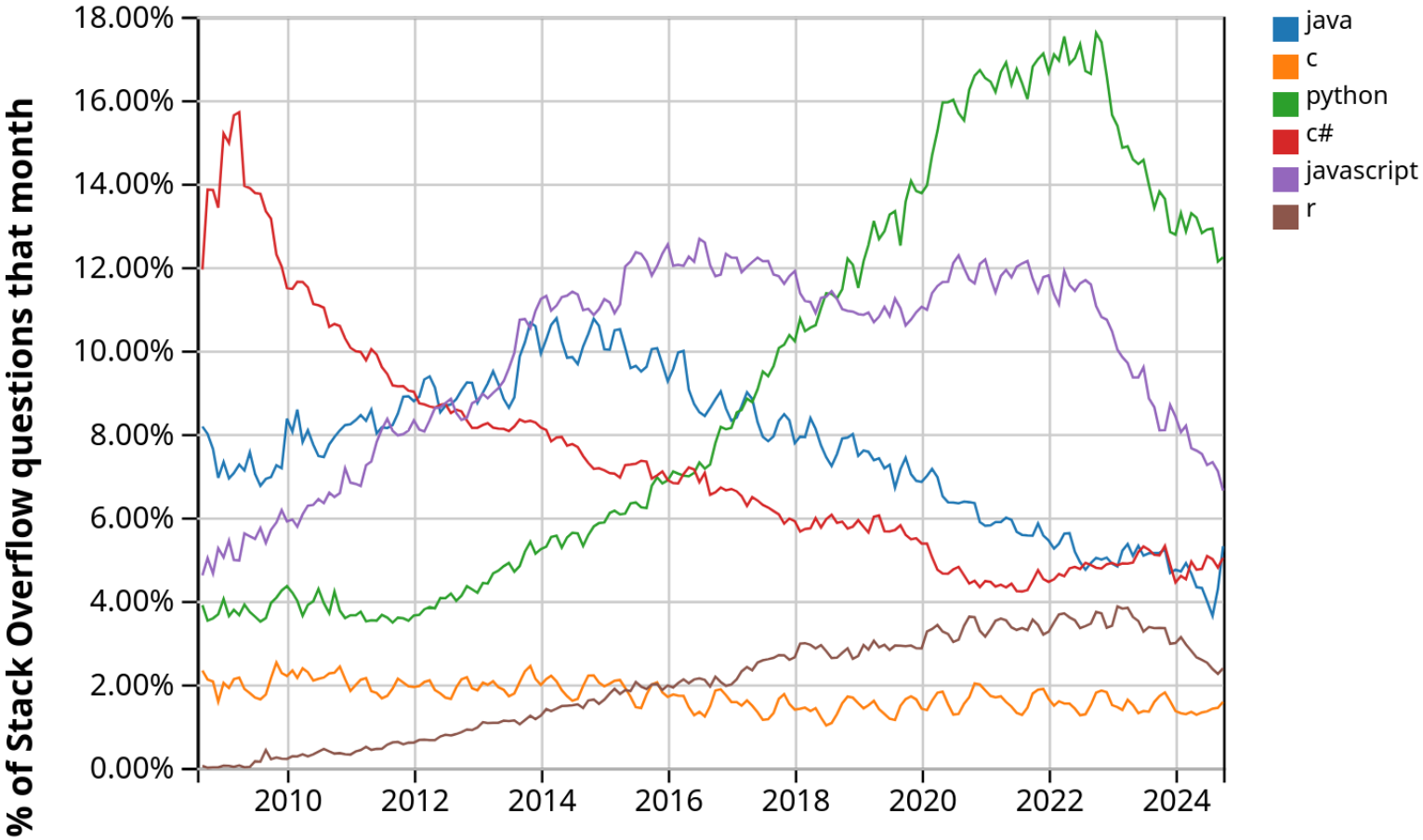
27 Jan 2025

The rules

1. Always feel free to search for help from us and neighbours! Questions? yes!
2. Teamwork! But everybody thinks AND types!
3. No one is "done" until everyone is (sort of) done!
4. But, do not simply provide solutions!

Why Python?

Why Python?


% of Stack Overflow questions that month
18.00%
16.00%
14.00%
12.00%
10.00%
8.00%
6.00%
4.00%
2.00%
0.00%
2010
2012
2014
2016
2018
2020
2022
2024
java
c
python
c#
javascript
r

Python

- General-purpose, object-oriented programming language
- Open-source and free
- 'Easy': high readability
- Fast (C++ backend for *numpy*)
- Great interactive environments
- Great support within the python community
  - packages, scripts, and example code
  - books and stackoverflow questions/replies
- Steep learning curve
- Independent of laptop, machine, operating system

Running python and Working environment

1. Run python line-by-line (console)

Running python and Working environment

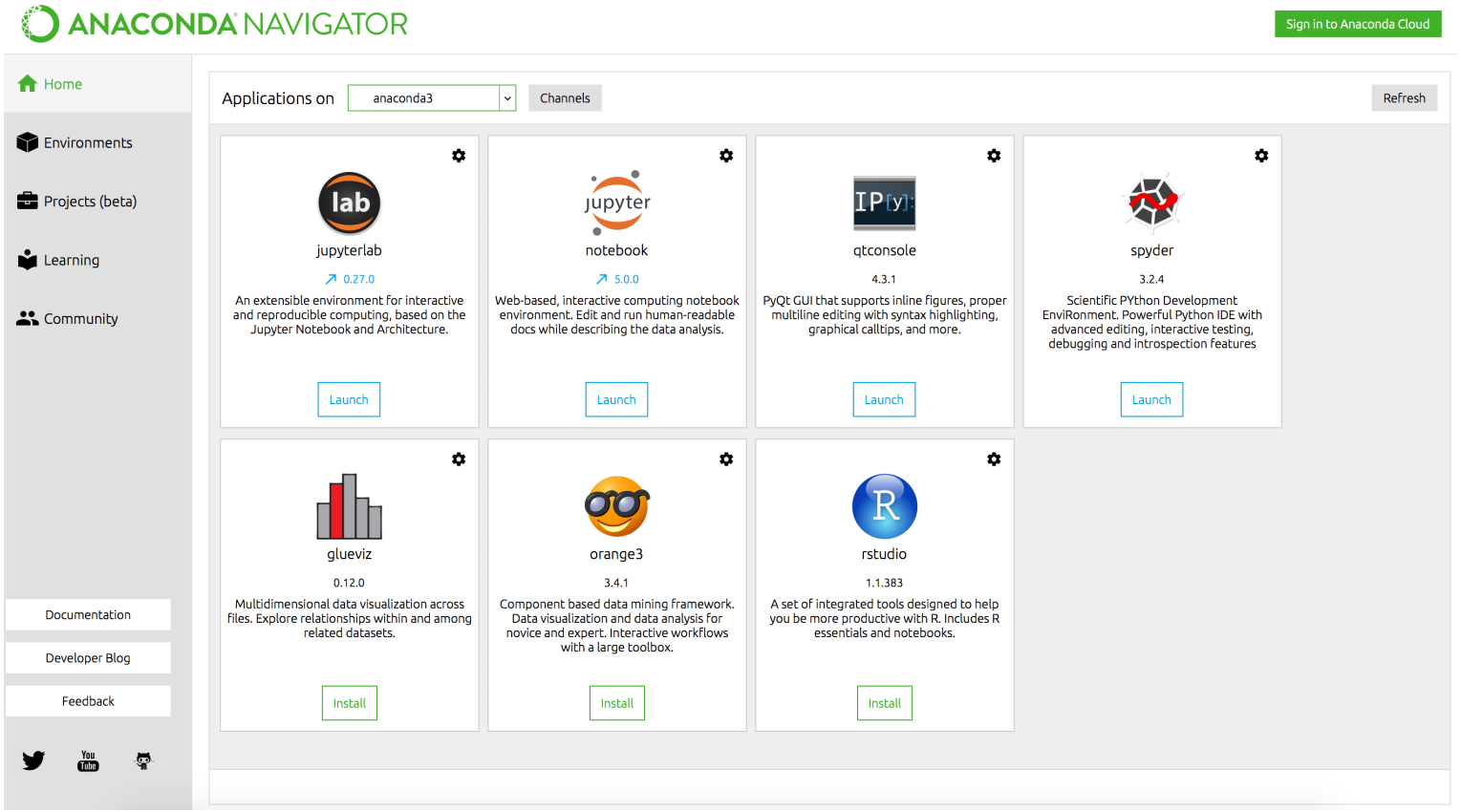
1. Run python line-by-line (console)
2. Create python scripts (e.g. *filename.py*) in text editor; execute via terminal/console: `python filename.py`

Running python and Working environment

1. Run python line-by-line (console)
2. Create python scripts (e.g. *filename.py*) in text editor; execute via terminal/console: `python filename.py`
3. Advanced text editors like *spyder*, vs code or jupyter notebooks (all available via Anaconda), which allow advanced functionalities like debugging, ...

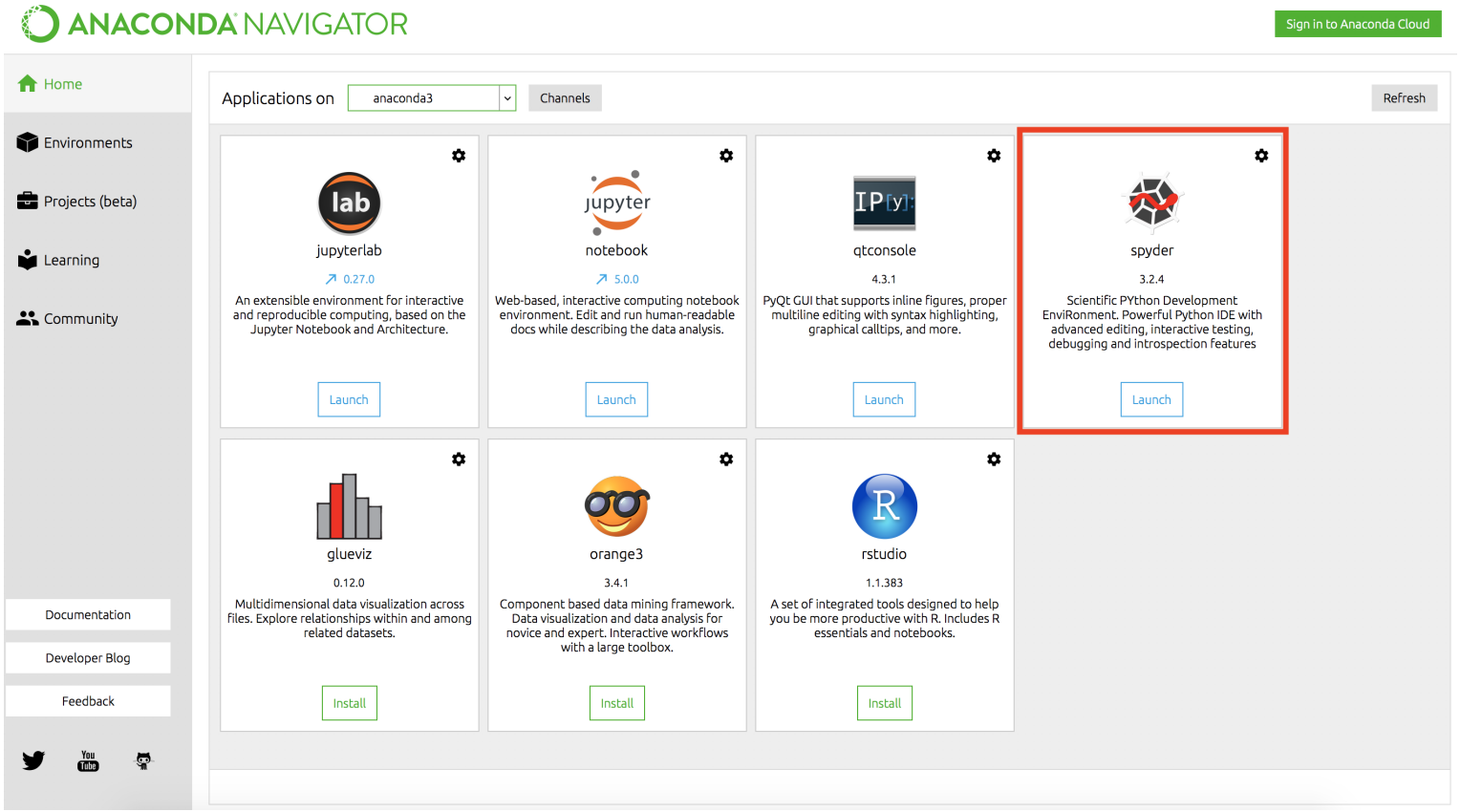
Anaconda distribution

Main anaconda navigator panel...



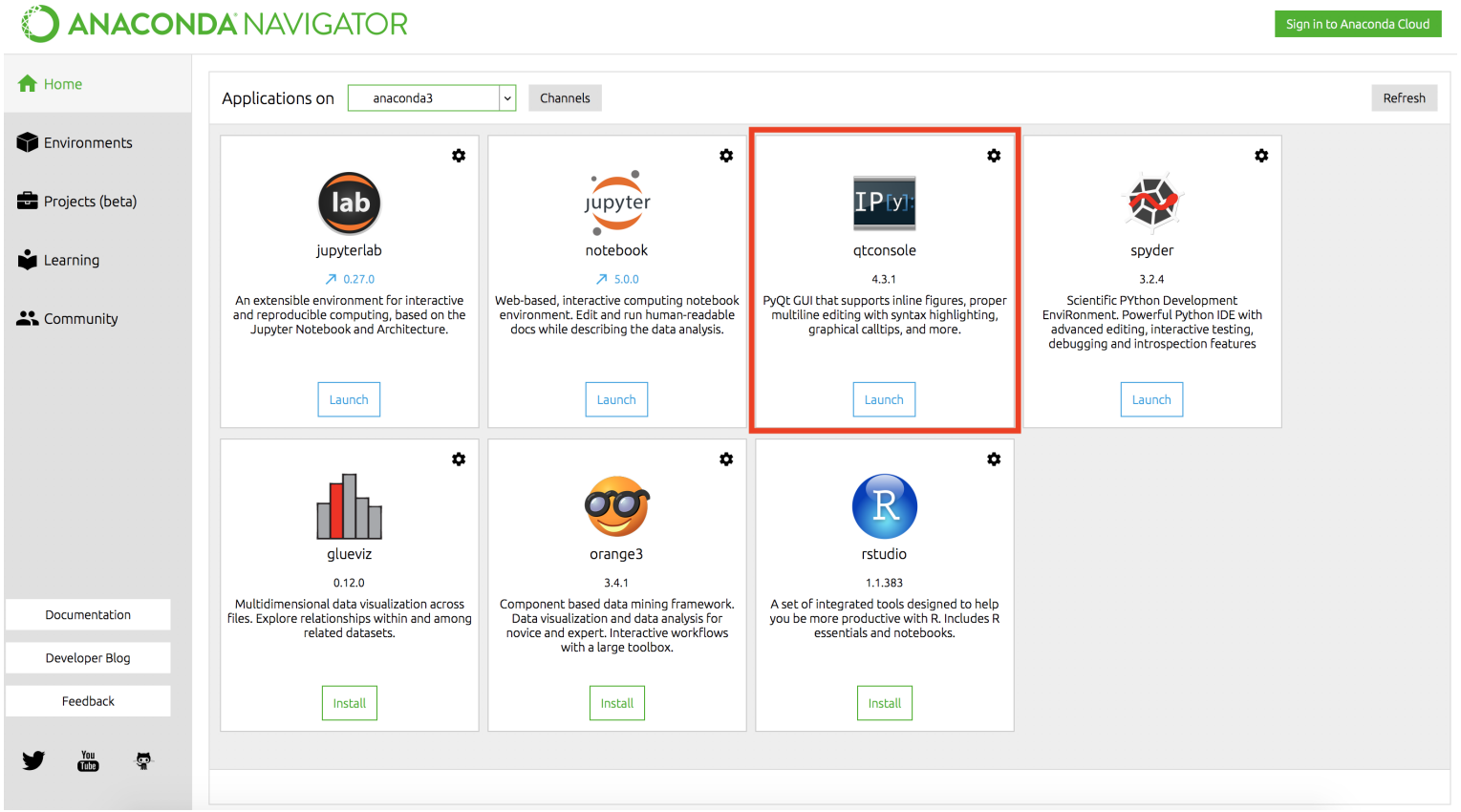
Anaconda distribution

Main anaconda navigator panel...



Anaconda distribution

Main anaconda navigator panel...



Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.

Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.
2. In this folder, create a file called README.txt and very briefly describe what is happening in this folder

Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.
2. In this folder, create a file called README.txt and very briefly describe what is happening in this folder
3. Also create folders *figs* and *data* (*data* not needed today)

Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.
2. In this folder, create a file called README.txt and very briefly describe what is happening in this folder
3. Also create folders *figs* and *data* (*data* not needed today)
4. Create a script called *helloworld.py*:

```
print("hello world!")
```

Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.
2. In this folder, create a file called README.txt and very briefly describe what is happening in this folder
3. Also create folders *figs* and *data* (*data* not needed today)
4. Create a script called *helloworld.py*:

```
print("hello world!")
```

5. Execute the file by navigating to the right folder in the terminal and running `python helloworld.py` (or click the run button in your advanced editor).

Workflow

1. Create a projcet subfolder, e.g. *day1_pythonbasics* in the winter school folder.
2. In this folder, create a file called README.txt and very briefly describe what is happening in this folder
3. Also create folders *figs* and *data* (*data* not needed today)
4. Create a script called *helloworld.py*:

```
print("hello world!")
```

5. Execute the file by navigating to the right folder in the terminal and running `python helloworld.py` (or click the run button in your advanced editor).

PLAYTIME

Familiarise yourself with these steps, create your first scripts (in this folder). Try math expressions, create variables, print results of math operations.

Overview

- Basic operations and variables
- Conditionals (if-else)
- Lists
- *Numpy* arrays
- Loops
- Defining functions
- Integration of ODEs with *scipy*'s function *odeint*
- Visualisation Basics with *matplotlib*
- ~~Dictionaries~~
- ~~Pandas DataFrames for Data Science~~
- ~~Statistical toolkits like scikit-learn~~

Basic operations

PLAYTIME

Familiarise yourself with these steps, create your first scripts (in this folder). Try math expressions, create variables, print results of math operations.

Basic operations

PLAYTIME

Familiarise yourself with these steps, create your first scripts (in this folder). Try math expressions, create variables, print results of math operations.

```
print("Hello world")
name = "Ago"  # a variable containing a string
print("Hello "+name)
a = 31
b = 11.5
c = a + b
print("the sum of ", a, " and ", b, " is ", c)
```

Basic operations

PLAYTIME

Familiarise yourself with these steps, create your first scripts (in this folder). Try math expressions, create variables, print results of math operations.

```
print("Hello world")
name = "Ago"  # a variable containing a string
print("Hello "+name)
a = 31
b = 11.5
c = a + b
print("the sum of ", a, " and ", b, " is ", c)
```

- Most operators work the way you expect it. Note, '2 to the power of 3' is `2**3`.
- Basic printing via `print(string1, variable1, ...)`
- Comments `#` (long comments: `"""a long comment..."""`, e.g. used for documentation string for a function)

Variable assignment

```
# some examples
a = 2  # defining a variable
b = 1
c = a+b
name = "Ago"  # defining a string variable
```

Variable assignment

```
# some examples
a = 2  # defining a variable
b = 1
c = a+b
name = "Ago"  # defining a string variable
```

- Declaring a variable in python means setting a *name* (left side of `=`, `c`) to hold a reference to some *object* (right side of the `=`, e.g. `a+b`).

Variable assignment

```
# some examples
a = 2  # defining a variable
b = 1
c = a+b
name = "Ago"  # defining a string variable
```

- Declaring a variable in python means setting a *name* (left side of `=`, `c`) to hold a reference to some *object* (right side of the `=`, e.g. `a+b`).
- Python figures our the variable type (next slide)

Variable assignment

```
# some examples
a = 2  # defining a variable
b = 1
c = a+b
name = "Ago"  # defining a string variable
```

- Declaring a variable in python means setting a *name* (left side of `=`, `c`) to hold a reference to some *object* (right side of the `=`, e.g. `a+b`).
- Python figures our the variable type (next slide)
- Variables can be overwritten or redefined.

Variable assignment

```
# some examples
a = 2  # defining a variable
b = 1
c = a+b
name = "Ago"  # defining a string variable
```

- Declaring a variable in python means setting a *name* (left side of `=`, `c`) to hold a reference to some *object* (right side of the `=`, e.g. `a+b`).
- Python figures our the variable type (next slide)
- Variables can be overwritten or redefined.
- Variable names can not have spaces!!! Allowed names: X, a, a_0, a0, last_name, ...

Basic data types

- Integer: `a = -5`

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`
- String: `c="hello"` — a string is a *list* of characters (see later)

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`
- String: `c="hello"` — a string is a *list* of characters (see later)
- Boolean `d=True` (or `False`)

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`
- String: `c="hello"` — a string is a *list* of characters (see later)
- Boolean `d=True` (or `False`)

What is?

- `a+b`? → a float (broadcasting)
- `a==b`?

- `a+d`?

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`
- String: `c="hello"` — a string is a *list* of characters (see later)
- Boolean `d=True` (or `False`)

What is?

- `a+b`? → a float (broadcasting)
- `a==b`? → a boolean
  (`==` is comparison, while `=` is variabel assignment)
- `a+d`?

Basic data types

- Integer: `a = -5`
- Float: `b=2.01`
- String: `c="hello"` — a string is a *list* of characters (see later)
- Boolean `d=True` (or `False`)

What is?

- `a+b`? → a float (broadcasting)
- `a==b`? → a boolean
  (`==` is comparison, while `=` is variabel assignment)
- `a+d`? → an integer; `True` is equal to `1`

Conditional statments (if-else)

Let's say we want to write an email and automate the greeting:

```
first_name = "Kwabena"
last_name = "Owusu"
friend = True
```

Desired output:

```
"Hey Kwabena"
```

```
first_name = "John"
last_name = "Mahama"
friend = False
```

Desired output:

```
"Dear Mr./Mrs. Mahama"
```

Conditional statments (if-else)

Let's say we want to write an email and automate the greeting:

```
first_name = "Kwabena"
last_name = "Owusu"
friend = True
```

Desired output:

```
"Hey Kwabena"
```

```
first_name = "John"
last_name = "Mahama"
friend = False
```

Desired output:

```
"Dear Mr./Mrs. Mahama"
```

Solution:

IF *condition* THEN *do block1* ELSE *do block2*

Conditional statments (if-else)

Let's say we want to write an email and automate the greeting:

```
first_name = "Kwabena"
last_name = "Owusu"
friend = True
```

Desired output:

```
"Hey Kwabena"
```

```
first_name = "John"
last_name = "Mahama"
friend = False
```

Desired output:

```
"Dear Mr./Mrs. Mahama"
```

Solution:

IF *condition* THEN *do block1* ELSE *do block2*

```
if friend==True:
  print("Hey ", first_name)
else:
  print("Dear ", "Mr./Mrs. ", last_name)
```

Conditional statments (if-else) II

```
if friend==True:
  print("Hey ", first_name)
else:
  print("Dear ", "Mr./Mrs. ", last_name)
```

Conditional statments (if-else) II

```
if friend==True:
  print("Hey ", first_name)
else:
  print("Dear ", "Mr./Mrs. ", last_name)
```

- Blocks and indentation. Semicolon ':' indicates the start of a block → Whitespace/Indentation matters!
- Note, assignment uses `=` and comparison uses `==`!
- Conditioncs can be combined through logical operators (`and`, `or`, `not`, `is not`) e.g.

  ```
  if (friend is False)and (first_name == "Kwabena"): ...
  ```

Conditional statments (if-else) II

```
if friend==True:
  print("Hey ", first_name)
else:
  print("Dear ", "Mr./Mrs. ", last_name)
```

- Blocks and indentation. Semicolon ':' indicates the start of a block → Whitespace/Indentation matters!
- Note, assignment uses `=` and comparison uses `==`!
- Conditioncs can be combined through logical operators (`and`, `or`, `not`, `is not`) e.g.

```
if (friend is False)and (first_name == "Kwabena"): ...
```

PLAYTIME

Create a file *ifelse.py* in your folder and re-create the script. Then include a binary gender attribute *gender*, which for now can be *male* or *female*, and determine whether the output should read "Mr." or "Mrs." accordingly.

Conditional statments (if-else) III

PLAYTIME

Create a file *ifelse.py* in your folder and re-create the script. Then include a binary gender attribute *gender*, which for now can be *male* or *female*, and determine whether the output should read "Mr." or "Mrs." accordingly.

Conditional statments (if-else) III

PLAYTIME

Create a file *ifelse.py* in your folder and re-create the script. Then include a binary gender attribute *gender*, which for now can be *male* or *female*, and determine whether the output should read "Mr." or "Mrs." accordingly.

```
first_name = "John"
last_name = "Mahama"
gender = "male"
friend = False
if friend:
  print("Hey ", first_name)
else:
  if gender == "female":
    print("Dear ", "Mrs. ", last_name)
  else:
    print("Dear ", "Mr. ", last_name)
```

- Note the levels of the indentation!

Lists

- Declare a list with square brackets and separate elements with commas:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
```

Lists

- Declare a list with square brackets and separate elements with commas:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
```

- Indexing: "give me the x-th element in the list". Counting is from the left and starts with index 0, 1, . . .:

```
names[0]
```

Lists

- Declare a list with square brackets and separate elements with commas:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
```

- Indexing: "give me the x-th element in the list". Counting is from the left and starts with index 0, 1, . . .:

```
names[0]
```

- Reassigning: A list is *mutable*, i.e. we can replace an element with a new one.

```
names[0] = "Kwabs"
```

Lists

- Declare a list with square brackets and separate elements with commas:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
```

- Indexing: "give me the x-th element in the list". Counting is from the left and starts with index 0, 1, . . .:

```
names[0]
```

- Reassigning: A list is *mutable*, i.e. we can replace an element with a new one.

```
names[0] = "Kwabs"
```

- In principle, a list can contain elements with different data types: `peter_attrs = [30, "green", "potato salad"]`

Lists

- Declare a list with square brackets and separate elements with commas:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
```

- Indexing: "give me the x-th element in the list". Counting is from the left and starts with index 0, 1, ...:

```
names[0]
```

- Reassigning: A list is *mutable*, i.e. we can replace an element with a new one.

```
names[0] = "Kwabs"
```

- In principle, a list can contain elements with different data types: `peter_attrs = [30, "green", "potato salad"]`

PLAYTIME

Create a file called *lists.py* and create a list of the nationalities of the five people next to you.

Lists II

- Negative indices:

  `names = ["Kwabena", "Ago", "Jospeh", "Peter"]`

  return the last element: `names[-1]`

  return the second last element: `names[-2]`

Lists II

- Negative indices:

  `names = ["Kwabena", "Ago", "Jospeh", "Peter"]`

  return the last element: `names[-1]`

  return the second last element: `names[-2]`

- Slicing: the ':' operator implies "from to"

  $\rightarrow$ to extract the first two elements of our list: `names[0:2]`

  What does `names[2:]` or `names[:-1]` do?

Lists II

- Negative indices:

  `names = ["Kwabena", "Ago", "Jospeh", "Peter"]`

  return the last element: `names[-1]`

  return the second last element: `names[-2]`
- Slicing: the ':' operator implies "from to"

  $\rightarrow$ to extract the first two elements of our list: `names[0:2]`

  What does `names[2:]` or `names[:-1]` do?
- Special commands/functions:
  - measure the length of a list: `len(names)`
  - index of an element (its first occurence): `names.index ('Ago')`
  - number of times an element occurs: `names.count("Joseph")`
  - test whether a value is inside a list: `"Kwabena"in names`
  - to append an element to a list: `names.append(["Vera"])` or simply `names + ["Vera"]`
  - special list: `range(start_num, stop_num, step)` e.g. `range(2,5,1)` means `2,3,4` (note `range(100)` means `0...99`).

Multidimensional lists

Lists can contain elements that are lists themselves:

```
a = [[1,2,3], [4,5,6], [7,8,9], [None, 0, None]]
```

What will be the result of `a[1][2]`?

Multidimensional lists

Lists can contain elements that are lists themselves:

```
a = [[1,2,3], [4,5,6], [7,8,9], [None, 0, None]]
```

What will be the result of `a[1][2]`?

`a[1]` gives us the second row (`[4,5,6]`), then `a[1][2]` gives us the digit 6.

Numpy — for everything related to math and numerical data

- Numpy arrays are the real workhorse of data structures for scientific applications

Numpy — for everything related to math and numerical data

- Numpy arrays are the real workhorse of data structures for scientific applications
- Array $\sim$ list but all elements are of the same type; here: floats

Numpy — for everything related to math and numerical data

- Numpy arrays are the real workhorse of data structures for scientific applications
- Array $\sim$ list but all elements are of the same type; here: floats
- But numpy allows FAST processing of a collection of numbers (C++ backend).

Numpy — for everything related to math and numerical data

- Numpy arrays are the real workhorse of data structures for scientific applications
- Array $\sim$ list but all elements are of the same type; here: floats
- But numpy allows FAST processing of a collection of numbers (C++ backend).
- Package *numpy* needs to be imported. We often use the shortcut *np*

```
import numpy as np
```

Creating a numpy array

- Creating one-dimensional arrays:
  1. from a list: `np.array([1,2,4,9])`

Creating a numpy array

- Creating one-dimensional arrays:
  1. from a list: `np.array([1,2,4,9])`
  2. using special functions:
     - `np.linspace(start, stop, num)`
     - `np.arange(start, stop, step)` (by default: `step=1`)
     - `np.ones(length)`
     - `np.zeros(length)`

Creating a numpy array

- Creating one-dimensional arrays:
  1. from a list: `np.array([1,2,4,9])`
  2. using special functions:
     - `np.linspace(start, stop, num)`
     - `np.arange(start, stop, step)` (by default: `step=1`)
     - `np.ones(length)`
     - `np.zeros(length)`
- Creating multi-dimensional arrays:
  1. list of lists: `np.array([[1,2], [3,4]])` → a 2x2 matrix
  2. functions, like: `np.zeros([5,5])` → a 5x5 matrix of zeros

Creating a numpy array

- Creating one-dimensional arrays:
  1. from a list: `np.array([1,2,4,9])`
  2. using special functions:
     - `np.linspace(start, stop, num)`
     - `np.arange(start, stop, step)` (by default: `step=1`)
     - `np.ones(length)`
     - `np.zeros(length)`
- Creating multi-dimensional arrays:
  1. list of lists: `np.array([[1,2], [3,4]])` → a 2x2 matrix
  2. functions, like: `np.zeros([5,5])` → a 5x5 matrix of zeros
  - Note, `arr.shape` returns the dimensions of a numpy array `arr`

Creating a numpy array

- Creating one-dimensional arrays:
  1. from a list: `np.array([1,2,4,9])`
  2. using special functions:
     - `np.linspace(start, stop, num)`
     - `np.arange(start, stop, step)` (by default: `step=1`)
     - `np.ones(length)`
     - `np.zeros(length)`
- Creating multi-dimensional arrays:
  1. list of lists: `np.array([[1,2], [3,4]])` → a 2x2 matrix
  2. functions, like: `np.zeros([5,5])` → a 5x5 matrix of zeros
  - Note, `arr.shape` returns the dimensions of a numpy array `arr`
- Indexing
  - accessing elements (indexing) is similar to lists: `arr[0]` returns first element, ...
  - for multi-dimensional array `arr`: `arr[1][2]` first row → `arr[1,2]` (row, column)

Numpy's *random* subpackage

Many of the models and simulations, we are interested in are stochastic and path-dependent.

```
import numpy as np
```

Numpy's *random* subpackage

Many of the models and simulations, we are interested in are stochastic and path-dependent.

```
import numpy as np
```

- a random number between 0 and 1:

  `np.random.random()`

  (can specify number of samples via function argument `size=...`)
- a random integer between 0 and 100:

  `np.random.randint(0,100)`
- a randomly sampled item from a list of options (with specific probabilities):

  `np.random.choice(<list-of-options>)`, e.g. `["A", "B", "C"]`

  (can specify probabilities and number of samples)

Numpy's *random* subpackage

Many of the models and simulations, we are interested in are stochastic and path-dependent.

```
import numpy as np
```

- a random number between 0 and 1:

  `np.random.random()`

  (can specify number of samples via function argument `size=...`)
- a random integer between 0 and 100:

  `np.random.randint(0,100)`
- a randomly sampled item from a list of options (with specific probabilities):

  `np.random.choice(<list-of-options>)`, e.g. `["A", "B", "C"]`

  (can specify probabilities and number of samples)
- Execute some action A with a certain probability p:

Numpy's *random* subpackage

Many of the models and simulations, we are interested in are stochastic and path-dependent.

```
import numpy as np
```

- a random number between 0 and 1:

  `np.random.random()`

  (can specify number of samples via function argument `size=...`)
- a random integer between 0 and 100:

  `np.random.randint(0,100)`
- a randomly sampled item from a list of options (with specific probabilities):

  `np.random.choice(<list-of-options>)`, e.g. `["A", "B", "C"]`

  (can specify probabilities and number of samples)
- Execute some action A with a certain probability p:
  Solution: `if np.random.random()<= p: A`

Operating with numpy arrays

```
arr = np.array([1,2,3,4])
```

Operating with numpy arrays

```
arr = np.array([1,2,3,4])
```

- Do math: `arr + 1` → `array([2,3,4,5])`, ...

Operating with numpy arrays

```
arr = np.array([1,2,3,4])
```

- Do math: `arr + 1` $\longrightarrow$ `array([2,3,4,5]), ...`
- Reassign/update entries: `arr[2] = 0`

Operating with numpy arrays

`arr = np.array([1,2,3,4])`

- Do math: `arr + 1` $\longrightarrow$ `array([2,3,4,5])`, ...
- Reassign/update entries: `arr[2] = 0`
- Special functions:
  - np.mean(arr)
  - np.max(arr), np.min(arr)
  - np.sum(arr)

Operating with numpy arrays

```
arr = np.array([1,2,3,4])
```

- Do math: `arr + 1` $\longrightarrow$ `array([2,3,4,5])`, ...
- Reassign/update entries: `arr[2] = 0`
- Special functions:
  - np.mean(arr)
  - np.max(arr), np.min(arr)
  - np.sum(arr)

PLAYTIME

Create a file *numpy-basics.py*. Careful: Do not call it *numpy.py*. Import the package, define two lists and convert them into numpy arrays. Then subtract the two arrays from each other.

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

```
def <name>(arg1, arg2, ... ,argN):
    <statements>
    return <value>
```

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

```
def <name>(arg1, arg2, ... ,argN):
    <statements>
    return <value>
```

- `def` creates a function and assigns it a name (one function, one name! There is no function overloading)

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

```
def <name>(arg1, arg2, ... ,argN):
    <statements>
    return <value>
```

- `def` creates a function and assigns it a name (one function, one name! There is no function overloading)
- `return` sends a result back to the caller. All functions have this! An empty (or no) return statement, returns the special value `None`.

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

```
def <name>(arg1, arg2, ... ,argN):
    <statements>
    return <value>
```

- `def` creates a function and assigns it a name (one function, one name! There is no function overloading)
- `return` sends a result back to the caller. All functions have this! An empty (or no) return statement, returns the special value `None`.
- `arguments` are passed to a function once it is called. They exist ONLY within the function.

Defining functions

Functions are helpful to structure, organise, and automate your code. If dealing with packages, they may be unavoidable.

```
def <name>(arg1, arg2, ... ,argN):
    <statements>
    return <value>
```

- `def` creates a function and assigns it a name (one function, one name! There is no function overloading)
- `return` sends a result back to the caller. All functions have this! An empty (or no) return statement, returns the special value `None`.
- `arguments` are passed to a function once it is called. They exist ONLY within the function.
- Optional arguments: In `def f(x, y, a=2)` the argument `a` does not need to be specified and defaults to $a = 2$.

Example Function and Function Call

Example function:

```
def times(x, y):
    """multiplies x with y"""
    z = x * y
    return z
```

Example Function and Function Call

Example function:

```
def times(x, y):
    """multiplies x with y"""
    z = x * y
    return z
```

Calling a function: `times(3, 4)` or more explicitly `times(x=3, y=4)`

Example Function and Function Call

Example function:

```
def times(x, y):
    """multiplies x with y"""
    z = x * y
    return z
```

Calling a function: `times(3, 4)` or more explicitly `times(x=3, y=4)`

PLAYTIME

Create a file called *functions.py*. Write a function that returns the value of some numerical operations. E.g. a function *square* that returns the square values of an array of numbers. Verify that your function works!

Loops

- For loops help to automate processes that are repeated:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
for name in names:
        print("Hello ", name)
```

Loops

- For loops help to automate processes that are repeated:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
for name in names:
        print("Hello ", name)
```

- → Structure: `for <new-variable> in <options>: ...`

Loops

- For loops help to automate processes that are repeated:

```
names = ["Kwabena", "Ago", "Jospeh", "Peter"]
for name in names:
        print("Hello ", name)
```

- → Structure: `for <new-variable> in <options>: ...`

PLAYTIME

Create a file *forloops.py*. Using a for loop, create a function that returns the mean value of a list of numbers (with unknown length).

PLAYTIME

Using a for loop again, create a function which takes a list of numbers as input and returns a list indicating which elements are even/odd (Tipp: check out `int(2.5)` and `int(2)`). E.g., `x=[1,4,2]` should return `even_number=[False, True, True]`

Additional stuff

- *While*-loop (rarely used): A *while* loop runs as long as some condition is fulfilled (careful, you may crash your laptop if it runs forever).

```
x = 1
while x<10:
  x = x*2
  print(x)
```

Additional stuff

- *While*-loop (rarely used): A *while* loop runs as long as some condition is fulfilled (careful, you may crash your laptop if it runs forever).

```
x = 1
while x<10:
  x = x*2
  print(x)
```

- *Enumerate* in combination with a list is sometimes very helpful:

```
for n, l in enumerate(["A", "B", "C"]):
  print("The letter, ", l, " has position ", n, " in
      the alphabet")
```

Scipy's differential equation integrator *odeint*

How to integrate a differential equation?
Traditionally, apply Newton-forward integration

```
y0 = 50  # initial condition

y = y0
def dydt(y, t):
  # exponential decay
  return -2 * y

# Integration via Forward-Euler
time_steps = range(100)
for t in time_steps:
  y = y + dydt(y, t) * dt
```

Scipy's differential equation integrator *odeint* II

odeint is an integrating tool in *scipy* that solves a system of ordinary differential equations (ODEs).
Instead of the Forward-Euler steps, we can write:

```
from scipy.integrate import odeint
y = odeint(dydt, y0, time_steps)
```

Scipy's differential equation integrator *odeint* II

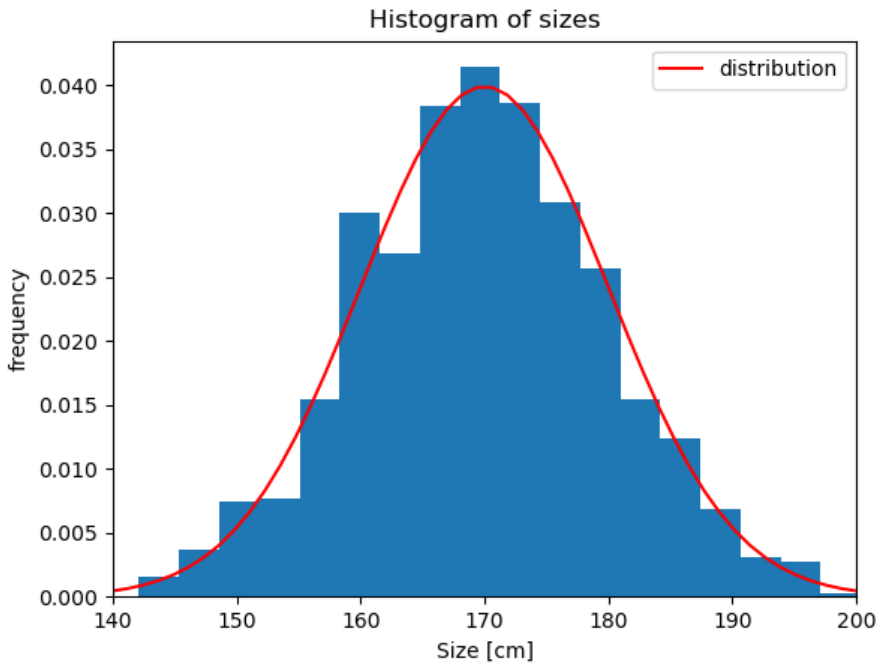
odeint is an integrating tool in *scipy* that solves a system of ordinary differential equations (ODEs).
Instead of the Forward-Euler steps, we can write:

```
from scipy.integrate import odeint
y = odeint(dydt, y0, time_steps)
```

For multi-dimensional ODEs, the initial condition (as well as the output of *dydt*) should be a *list*, where each element of the list represents a variabel (and its derivative).

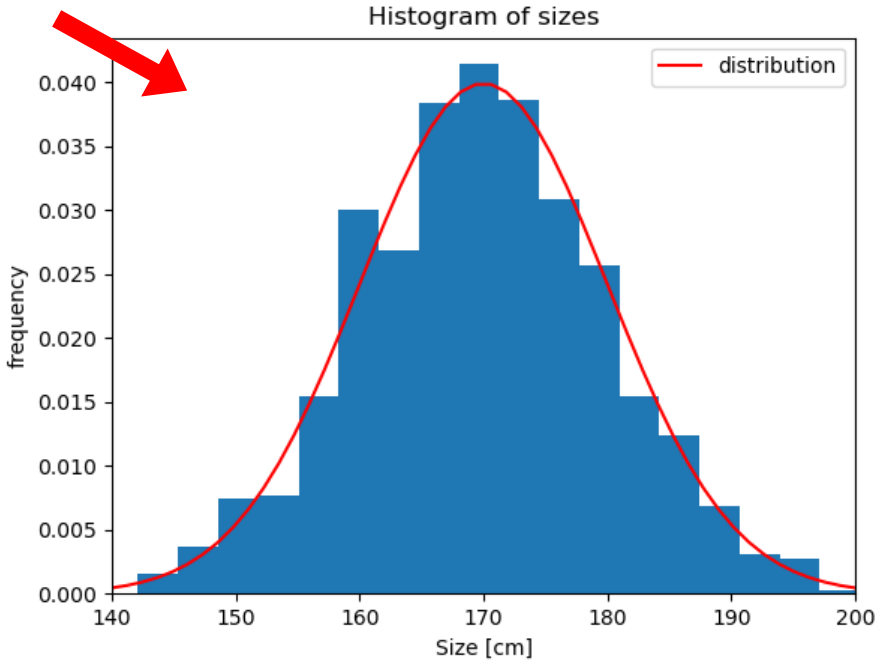
Visualisation with *matplotlib*

```python
import numpy as np
import matplotlib.pyplot as plt

# create samples from a normal distribution with mean=100 and standard deviation=15
samples = np.random.normal(170, 10, size=1000)

# create the distribution
x = np.linspace(140, 200)
y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))

fig = plt.figure()
ax = plt.axes()

ax.hist(samples, bins=20, density=True)
ax.plot(x, y, color="red", label="distribution")
ax.set_xlim(140,200)
ax.set_ylim(0,)
ax.legend()
ax.set_xlabel("Size [cm]")
ax.set_ylabel("frequency")
ax.set_title("Histogram of sizes")
plt.show()

```



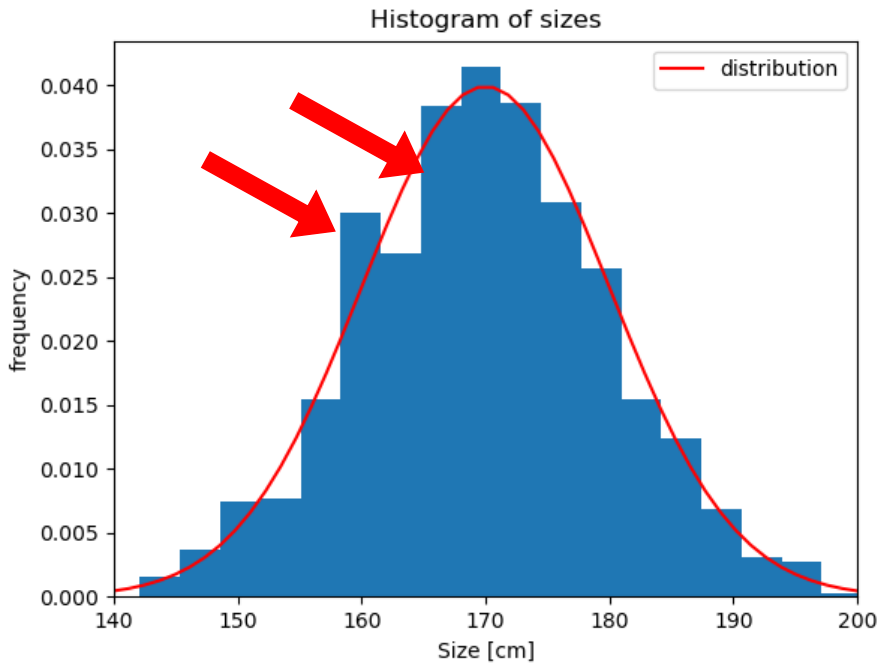
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



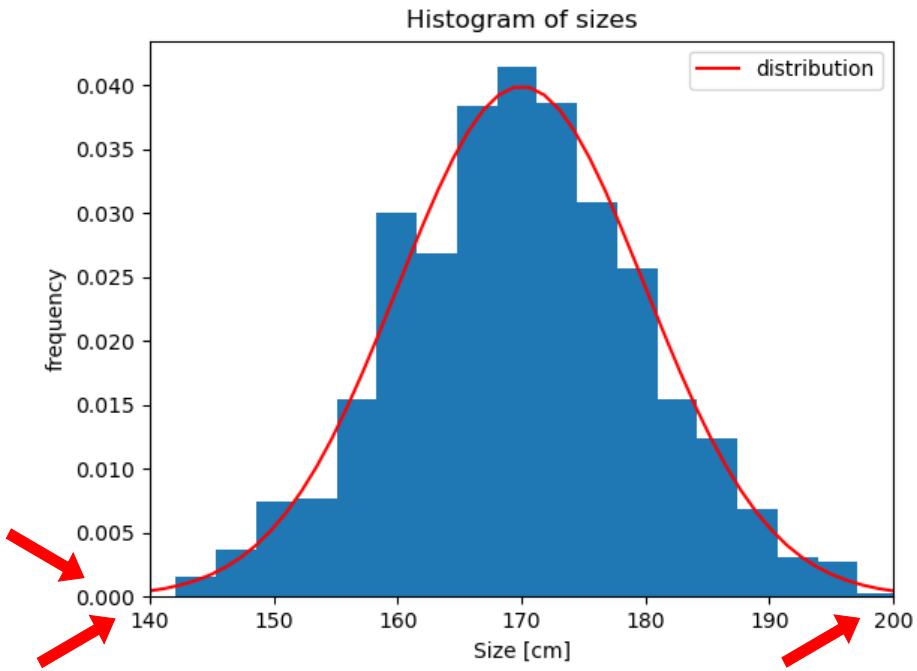
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



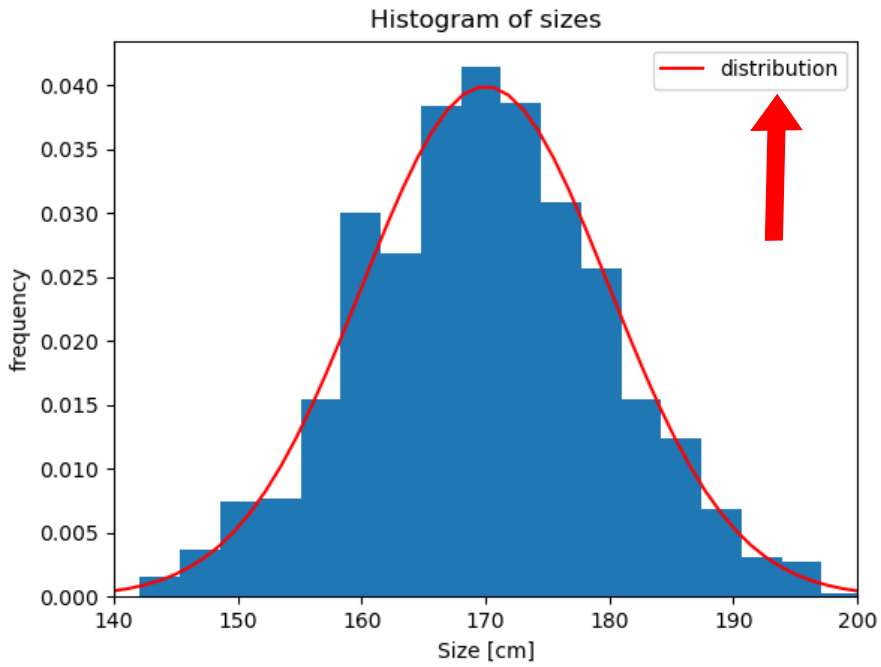
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



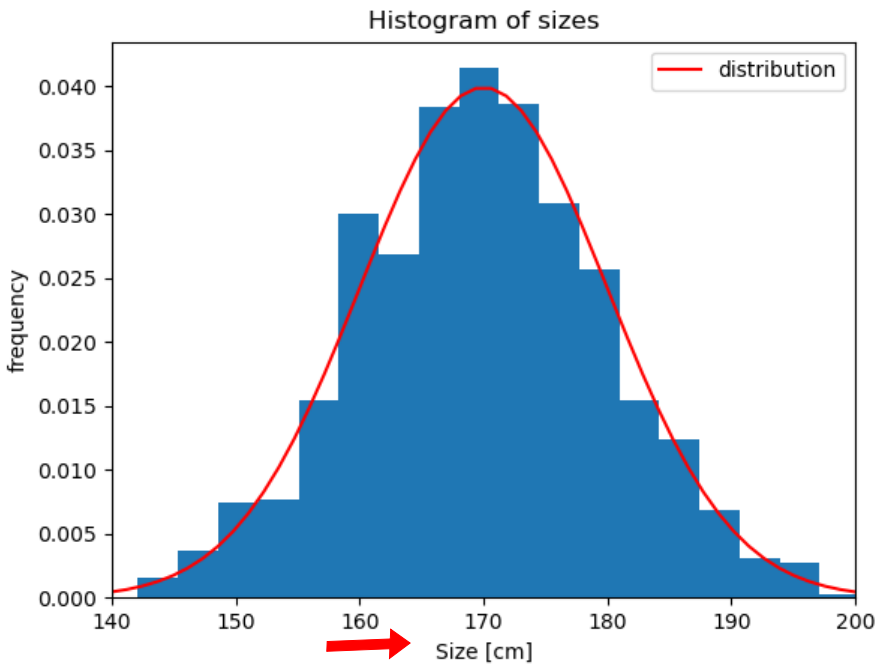
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



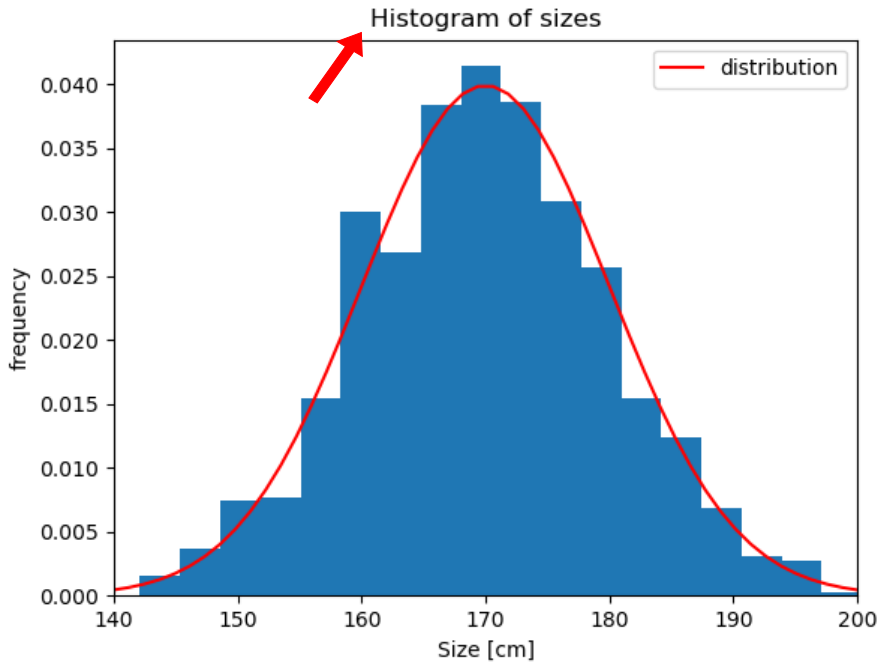
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



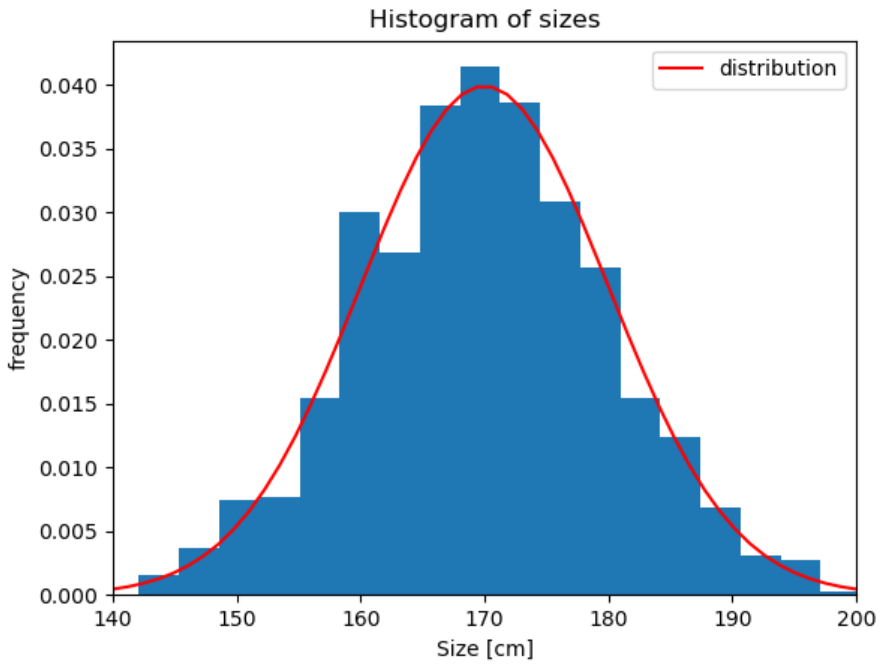
Visualisation with *matplotlib*

```python
import numpy as np
import matplotlib.pyplot as plt

# create samples from a normal distribution with mean=100 and
standard deviation=15
samples = np.random.normal(170, 10, size=1000)

# create the distribution
x = np.linspace(140, 200)
y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))

fig = plt.figure()
ax = plt.axes()

ax.hist(samples, bins=20, density=True)
ax.plot(x, y, color="red", label="distribution")
ax.set_xlim(140,200)
ax.set_ylim(0,)
ax.legend()
ax.set_xlabel("Size [cm]")
ax.set_ylabel("frequency")
ax.set_title("Histogram of sizes")
plt.show()

```



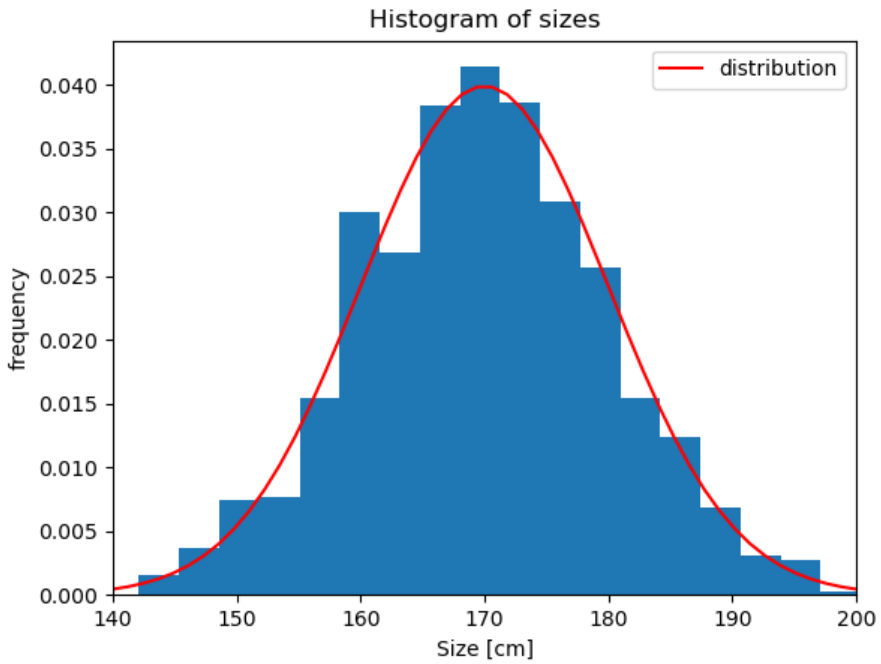
Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



Visualisation with *matplotlib*

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # create samples from a normal distribution with mean=100 and
    standard deviation=15
5   samples = np.random.normal(170, 10, size=1000)
6
7   # create the distribution
8   x = np.linspace(140, 200)
9   y = 1/(2*np.pi*10**2)**0.5 * np.exp(-(x-170)**2/(2 * 10**2))
10
11  fig = plt.figure()
12  ax = plt.axes()
13
14  ax.hist(samples, bins=20, density=True)
15  ax.plot(x, y, color="red", label="distribution")
16  ax.set_xlim(140,200)
17  ax.set_ylim(0,)
18  ax.legend()
19  ax.set_xlabel("Size [cm]")
20  ax.set_ylabel("frequency")
21  ax.set_title("Histogram of sizes")
22  plt.show()
23
```



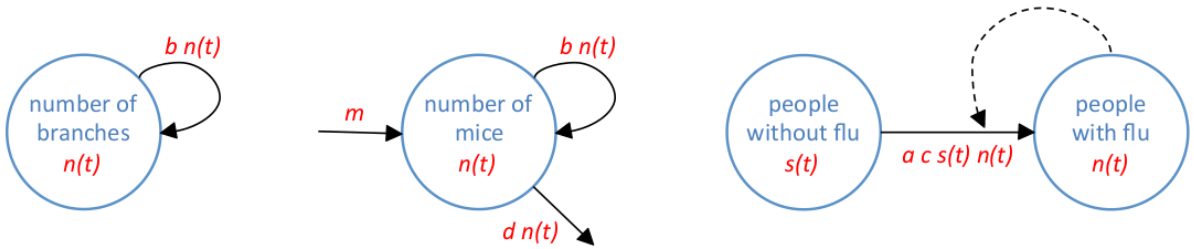
save the plot: `plt.savefig(figs/histogram.pdf)`.

Overview

- Basic operations and variables
- Conditionals (if-else)
- Lists
- *Numpy* arrays
- Loops
- Defining functions
- Integration of ODEs with *scipy*'s function *odeint*
- Visualisation Basics with *matplotlib*
- ~~Dictionaries~~
- ~~Pandas DataFrames for Data Science~~
- ~~Statistical toolkits like scikit-learn~~

Break

First basic models



Branching

1.1 Code the 'branching' model, i.e.: $dn(t)/dt = b \cdot n(t)$

1.2 Use an initial number of branches of $n(t = 0) = 10.0$ and a branching rate of $b = 0.25$ branches per day

1.3 Generate a solution $n(t)$ over 101 evenly spaced time steps over 10 days

1.4 Plot the results.

Cat and mouse

2.1 Code the 'cat and mouse' model, i.e.: $dn(t)/dt = b \cdot n(t) - d \cdot n(t) + m$

2.2 Use an initial number of mice of $n(t = 0) = 10.0$, a birth rate of 2 mice every 8 days, each day the cat decreases the mice population to 1/2 of its value, and 3 new mice migrate every 10 days from nearby yards

2.3 Generate a solution $n(t)$ over 101 evenly spaced time steps over 10 days

2.4 Plot the results.

Flu

3.1 Code the 'flu' model, i.e.: $ds(t)/dt = -a \cdot c \cdot n(t) \cdot s(t)$ and $dn(t)/dt = +a \cdot c \cdot s(t) \cdot n(t)$

3.2 Assume 10 contacts between carrier and healthy individuals every day and a 2.5 % probability that the flu is transmitted at every contact

3.3 Use an initial number of healthy individuals of $s(t = 0) = 95.0$, an initial number of carriers of $n(t = 0) = 5.0$

3.4 Generate a solution over 101 evenly spaced time steps over 10 days

3.5 Plot the results

3.6 Plot both $s(t)$ and $n(t)$ and $s(t) + n(t)$ (i.e. the total number of individuals) in the same panel; after how many days does the model reach steady-state?

A plotting solution for the branching model

Consider `t_array = np.linspace(0,10,101)` and `y` being the solution of the `odeint`-call.

```
fig = plt.figure()
ax = plt.axes()
ax.plot(t_array, y, color="k", label="branching with rate b="+str(b))
plt.legend()
ax.set_xlabel("time t")
ax.set_ylabel("variable y")
plt.show()
# plt.savefig("figs/branchingModel.png")
```