A Practical Introduction to Machine Learning in Python
Day 1 - Monday Morning
»Introduction«

Rupert Kiddle
Marieke van Hoof

r.t.kiddle@vu.nl
m.vanhoof@uva.nl

September 16, 2024

Gesis

Today

- Introducing...
  - ...the people
- Python: A language, not a program
  - A good workflow
  - Clean, high-quality code
  - Exercise
  - datatypes
  - Generators
  - Scaling up
  - data storage
- Looking forward
  - And now you

All course materials can be found at...

https://github.com/annekroon/gesis-machine-learning/fall-2024

Introducing. . .

Introducing. . .

. . . the people

Introducing...

...the people

Introducing...

Rupert Kiddle

PhD Candidate: Computational Communication Science (RM SocSci, BSc PolSci, BSc Neurosci)

- Vrije University Amsterdam, Journalism
- interested in (digital) journalism, language modelling and news recommendation.

@rptkiddle |r.t.kiddle@vu.nl

Introducing...

Marieke van Hoof
Postdoctoral researcher AI & Politics (PhD PolCom, RM SocSci, BSc Sociology)

- Amsterdam School of Communication Research, University of Amsterdam
- Research focused on the role of AI and algorithms in political information diets
- Traditional methods (e.g., surveys, experiments) & text analysis using automated approaches

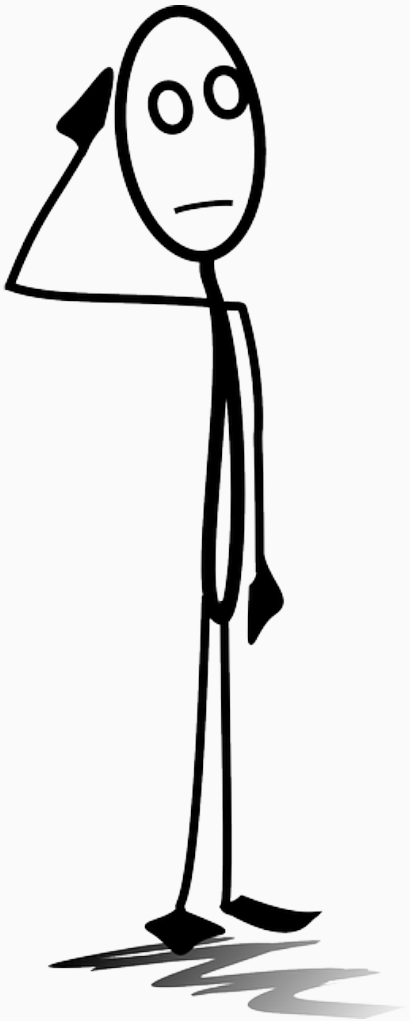
@marieke_vh |m.vanhoof@uva.nl

Introducing...

prof. dr. Damian Trilling
Vrije Universiteit Amsterdam

dr. Anne Kroon
University of Amsterdam

Introducing...



Your name?

Your background?

Your reason to follow this course?

Do you have a dataset you are working on?

Short poll

Do you need

a. an intro
b. a brief refresher
c. nothing

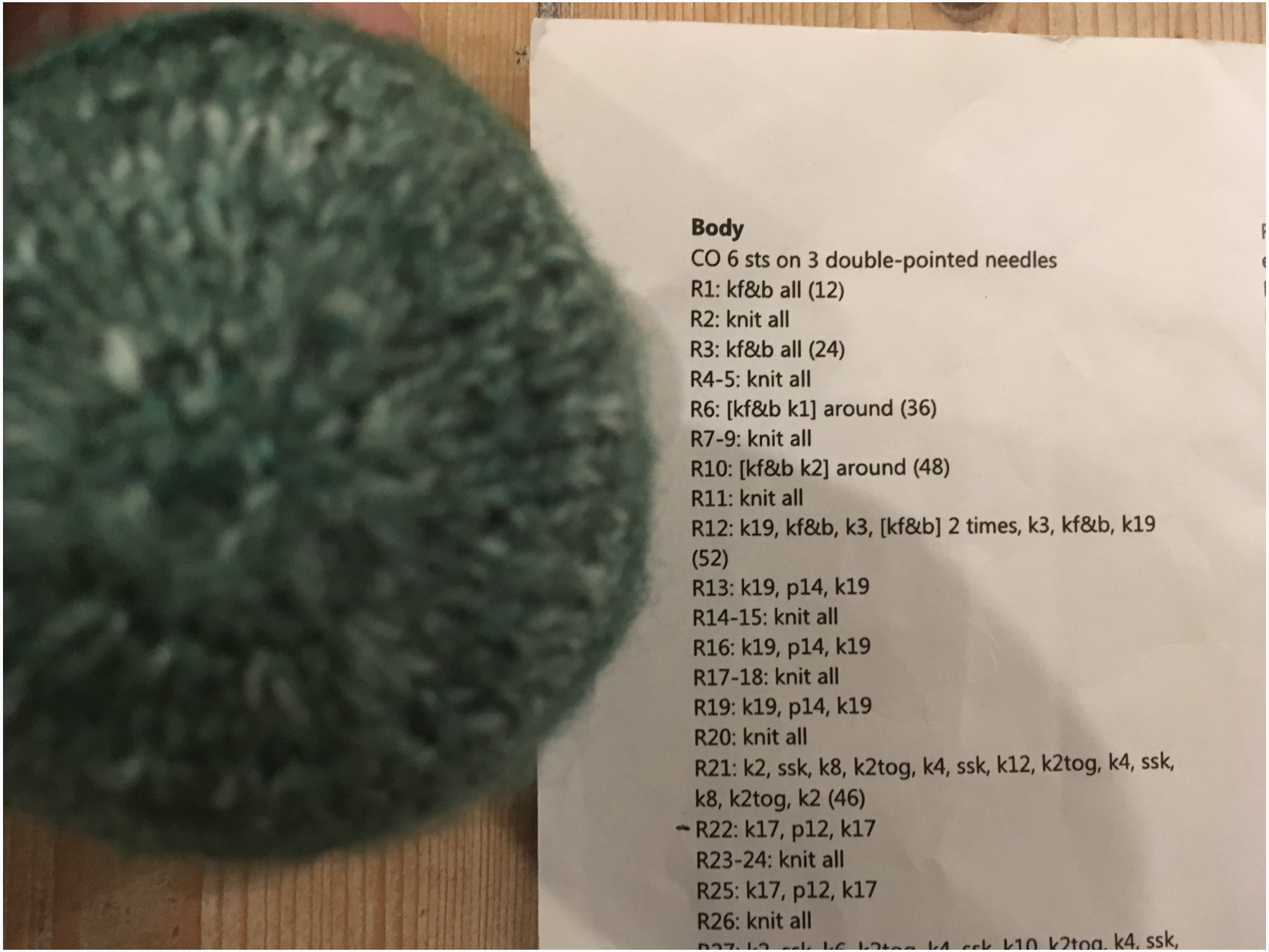
on

i. datatypes (int, float, string, lists, dictionaries)
ii. control flow statements (for, if, try/except)
iii. ways to run your code (notebooks vs IDE's vs text editors)

?

We will try do adapt today's programme to your needs!

Python: A language, not a program



An algorithm in a language that's a bit harder (I think) than Python

Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)
  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We will use 3.10. Generally a good idea to use a recent version, but not the very latest.

Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)
  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We will use 3.10. Generally a good idea to use a recent version, but not the very latest.

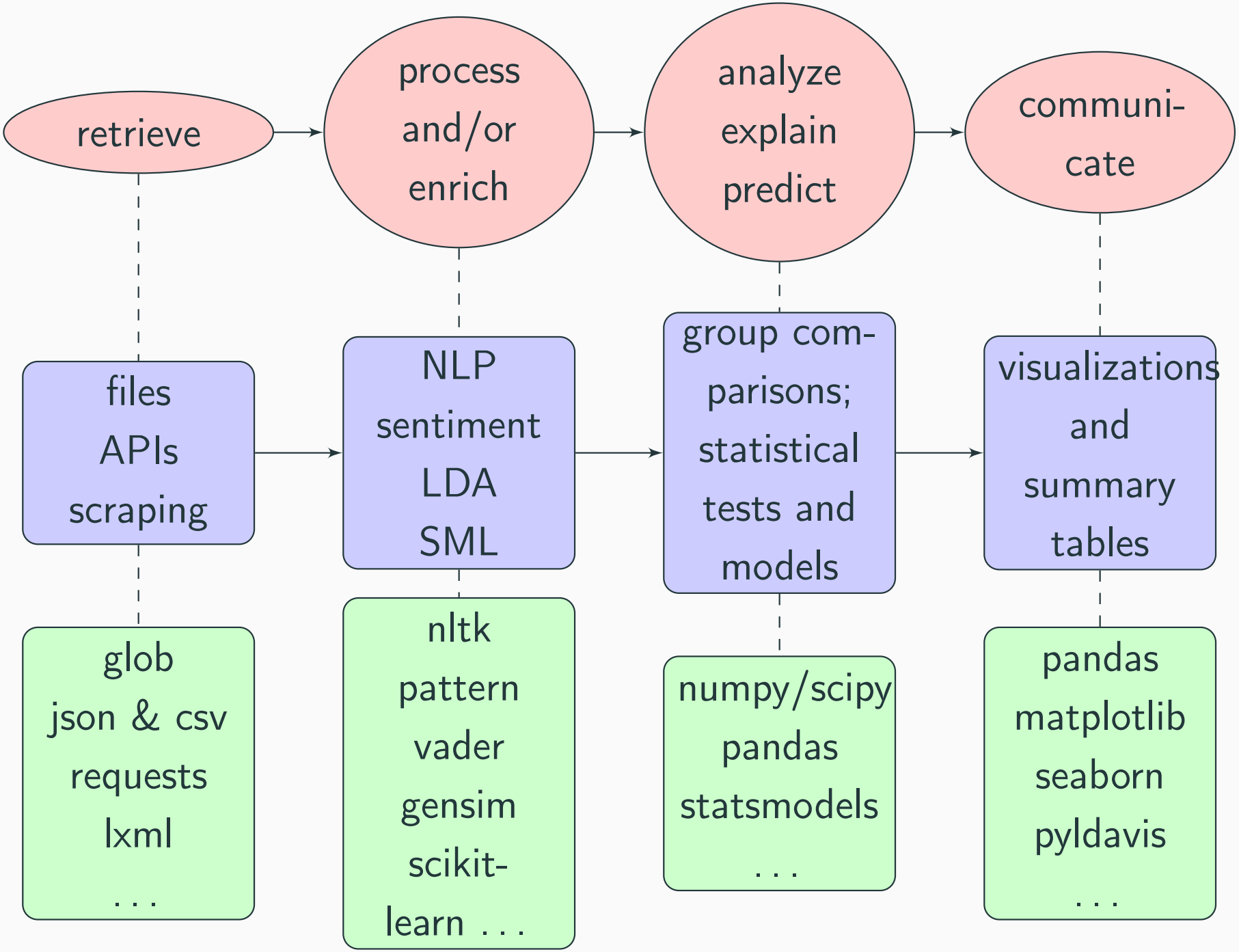
Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)
  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We will use 3.10. Generally a good idea to use a recent version, but not the very latest.


retrieve
process and/or enrich
analyze explain predict
communi-cate
files APIs scraping
NLP sentiment LDA SML
group com-parisons; statistical tests and models
visualizations and summary tables
glob json & csv requests lxml ...
nltk pattern vader gensim scikit-learn ...
numpy/scipy pandas statsmodels ...
pandas matplotlib seaborn pyldavis ...

Python: A language, not a program

A good workflow

A good workflow

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

- Reusable data and code
- Efficiency and productivity
- Reorganization of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

- Reusable data and code
- Efficiency and reliability
- Reorganization of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

1. Reusable data and code
2. Efficiency and productivity
3. Recognition of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

1. Reusable data and code
2. Efficiency and productivity
3. Recognition of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

- Reusable data and code
- Efficiency and credibility
- Recognition of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

- Reusable data and code
- Efficiency and credibility
- Recognition of tools and data

Maximize transparency

Maximizing transparency of code and data

- Use openly accessible repository (e.g., Github)
- Store and preserve (pseudonymised) data at a secure environment (e.g., OSF)
- Create reusable workflows

Advantages

- Reusable data and code
- Efficiency and credibility
- Recognition of tools and data

Python: A language, not a program

Clean, high-quality code

Develop components separately

One script for downloading the data, one script for analyzing

- Avoids waste of resources (e.g., unnecessary downloading multiple times)
- Makes it easier to re-use your code or apply it to other data

Start small, then scale up

- Take your plan and solve one problem at a time (e.g. parsing one review page, or getting the HTML of a review page)
- (for instance, by using functions (next slide))

Develop components separately

One script for downloading the data, one script for analyzing

- Avoids waste of resources (e.g., unnecessary downloading multiple times)
- Makes it easier to re-use your code or apply it to other data

Start small, then scale up

- Take your plan and solve one problem at a time (e.g., parsing one review page, or getting the HTML of one review page)
- (for instance, by using functions (next slide))

Develop components separately

One script for downloading the data, one script for analyzing

- Avoids waste of resources (e.g., unnecessary downloading multiple times)
- Makes it easier to re-use your code or apply it to other data

Start small, then scale up

- Take your plan and solve one problem at a time (e.g., parsing one review page, or getting the HTML of one review page)
- (for instance, by using functions (next slide))

Develop components separately

One script for downloading the data, one script for analyzing

- Avoids waste of resources (e.g., unnecessary downloading multiple times)
- Makes it easier to re-use your code or apply it to other data

Start small, then scale up

- Take your plan and solve *one* problem at a time (e.g., parsing a review page; or getting the URLs of all review pages)
- (for instance, by using functions [next slides])

Develop components separately

One script for downloading the data, one script for analyzing

- Avoids waste of resources (e.g., unnecessary downloading multiple times)
- Makes it easier to re-use your code or apply it to other data

Start small, then scale up

- Take your plan and solve *one* problem at a time (e.g., parsing a review page; or getting the URLs of all review pages)
- (for instance, by using functions [next slides])

Develop components separately

If you copy-paste code, you are doing something wrong

- Write loops!
- If something takes more than a couple of lines, write a function!

Develop components separately

If you copy-paste code, you are doing something wrong

- Write loops!
- If something takes more than a couple of lines, write a function!

Copy-paste approach
(ugly, error-prone, hard to scale up)

```
allreviews = []

response = requests.get('http://xxxxx')
tree = fromstring(response.text)
reviewelements = tree.xpath('//div[@class="review"]')
reviews = [e.text for e in reviewelements]
allreviews.extend(reviews)

response = requests.get('http://yyyyy')
tree = fromstring(response.text)
reviewelements = tree.xpath('//div[@class="review"]')
reviews = [e.text for e in reviewelements]
allreviews.extend(reviews)
```

Better: for-loop

(easier to read, less error-prone, easier to scale up (e.g., more URLs, read URLs from a file or existing list)

```
allreviews = []

urls = [‘http://xxxxx’, ‘http://yyyyy’]

for url in urls:
    response = requests.get(url)
    tree = fromstring(response.text)
    reviewelements = tree.xpath(‘//div[@class="review"]’)
    reviews = [e.text for e in reviewelements]
    allreviews.extend(reviews)
```

Even better: for-loop with functions

(main loop is easier to read, function can be re-used in multiple contexts)

```
1  def getreviews(url):
2      response = requests.get(url)
3      tree = fromstring(response.text)
4      reviewelements = tree.xpath('//div[@class="review"]')
5      return [e.text for e in reviewelements]
6
7
8  urls = ['http://xxxxx', 'http://yyyyy']
9
10 allreviews = []
11
12 for url in urls:
13     allreviews.extend(getreviews(url))
```

Python: A language, not a program

Exercise

Exercises

Did you pass?

- Think of a way to determine for a list of grades whether they are a pass (>5.5) or fail.
- Can you make that program robust enough to handle invalid input (e.g., a grade as 'ewghjieh')?
- How does your program deal with impossible grades (e.g., 12 or -3)?
- ...

Python: A language, not a program

datatypes

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome to write operations on
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome, e.g. when passing parameters
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome, e.g. when passing parameters
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome (e.g. no named tuples)
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome - e.g. no enforcement of structure
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- Can be more cumbersome - e.g. no enforcement of structure
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- can be more cumbersome: e.g., inserting a column
- less consistency checks

Datatypes

Low-level: Native python datatypes

- Booleans, integers, floats, strings, bytes, byte arrays
- Lists, typles, sets, dictionaries

Advantages

- fast, flexible
- allows for nested, unstructured data

Disadvantages

- can be more cumbersome: e.g., inserting a column
- less consistency checks

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively for tabular data
- easy, allows for pretty visualization

Disadvantages

- not suited for one-line tasks or messy / exploratory tasks
- when your data is very large (machine learning)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively for tabular data
- easy, allows for pretty visualization

Disadvantages

- not suited for one-line tasks or messy / exploratory tasks
- when your data is very large (machine learning)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively (for tabular data)
- easy, allows for pretty visualization

Disadvantages

- not suited for one-dimensional or messy / deeply nested data
- when your data is very large (machine learning!)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively (for tabular data)
- easy, allows for pretty visualization

Disadvantages

- not suited for one-dimensional or messy / deeply nested data
- when your data is very large (machine learning!)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively (for tabular data)
- easy, allows for pretty visualization

Disadvantages

- not suited for one-dimensional or messy / deeply nested data
- when your data is very large (machine learning!)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively (for tabular data)
- easy, allows for pretty visualization

Disadvantages

- not suited for one-dimensional or messy / deeply nested data
- when your data is very large (machine learning!!)

Datatypes

Higher-level: importing modules

- e.g., numpy, pandas, seaborn

Advantages

- useful convenience functionality, works very intuitively (for tabular data)
- easy, allows for pretty visualization

Disadvantages

- not suited for one-dimensional or messy / deeply nested data
- when your data is very large (machine learning!!)

Datatypes in this course

In this week, we will mainly work with lower-level datatypes (as opposed to, for instance, pandas dataframes)

- Often, ML algorithms require native data types as input (i.e., lists, genertors)
- We have to seriously consider memory:
- Maybe size does not apply to your project yet, but in the future you might want to scale up.

Datatypes in this course

In this week, we will mainly work with lower-level datatypes (as opposed to, for instance, pandas dataframes)

- Often, ML algorithms require native data types as input (i.e., lists, genertors)
- We have to seriously consider memory:
- Maybe size does not apply to your project yet, but in the future you might want to scale up.

Datatypes in this course

In this week, we will mainly work with lower-level datatypes (as opposed to, for instance, pandas dataframes)

- Often, ML algorithms require native data types as input (i.e., lists, genertors)
- We have to seriously consider memory:
- Maybe size does not apply to your project yet, but in the future you might want to scale up.

Python: A language, not a program

Generators

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

1. Does not hold results in memory
2. Only computes results at the moment you need them (be lazy)
3. You can only loop over your object ONCE

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

- Does not hold results in memory
- Only computes results at the moment you need them (lazy)
- You can only loop over your object once

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

- Does not hold results in memory
- Only computes results at the moment you need them (lazy)
- You can only loop over your object once

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

- Does not hold results in memory
- Only computes results at the moment you need them (i.e. lazy')
- You can only loop over your object ONCE.

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

- Does not hold results in memory
- Only computes results at the moment you need them (i.e. lazy')
- You can only loop over your object ONCE.

Generators

Generators

- We will work with *generators* to deal with memory issues
- Generators behave like iterators: loops through elements of an object.

Behavior of a generator

- Does not hold results in memory
- Only computes results at the moment you need them (i.e. lazy')
- You can only loop over your object ONCE.

Creating generators: Example 1

```
def my_generator(my_list):
    for i in my_list:
        yield i
example_list = [1, 2, 3, 4]
gen1 = my_generator(example_list)
next(gen1)
```

This will return:

```
1
```

Creating generators: Example 2 (shorter)

```
1  my_list = [1,2,3,4]
2  gen = (i for i in my_list)
```

Python: A language, not a program

Scaling up

Scaling up

When considering datatypes, consider re-usability, scalability

- Use functions and classes to make code more readable and re-usable
- Avoid re-calculating values
- Think about how to minimize memory usage (e.g., Generators)
- Do not hard-code values, file names, etc., but take them as arguments

Make it robust

You cannot foresee every possible problem.
Most important: Make sure your program does not fail and loose all data just because something goes wrong at case 997/1000.

- Use `try/except` to explicitly tell the program how to handle errors
- Write data to files (or database) in between
- Use `assert len(x) == len(y)` for sanity checks

Python: A language, not a program

data storage

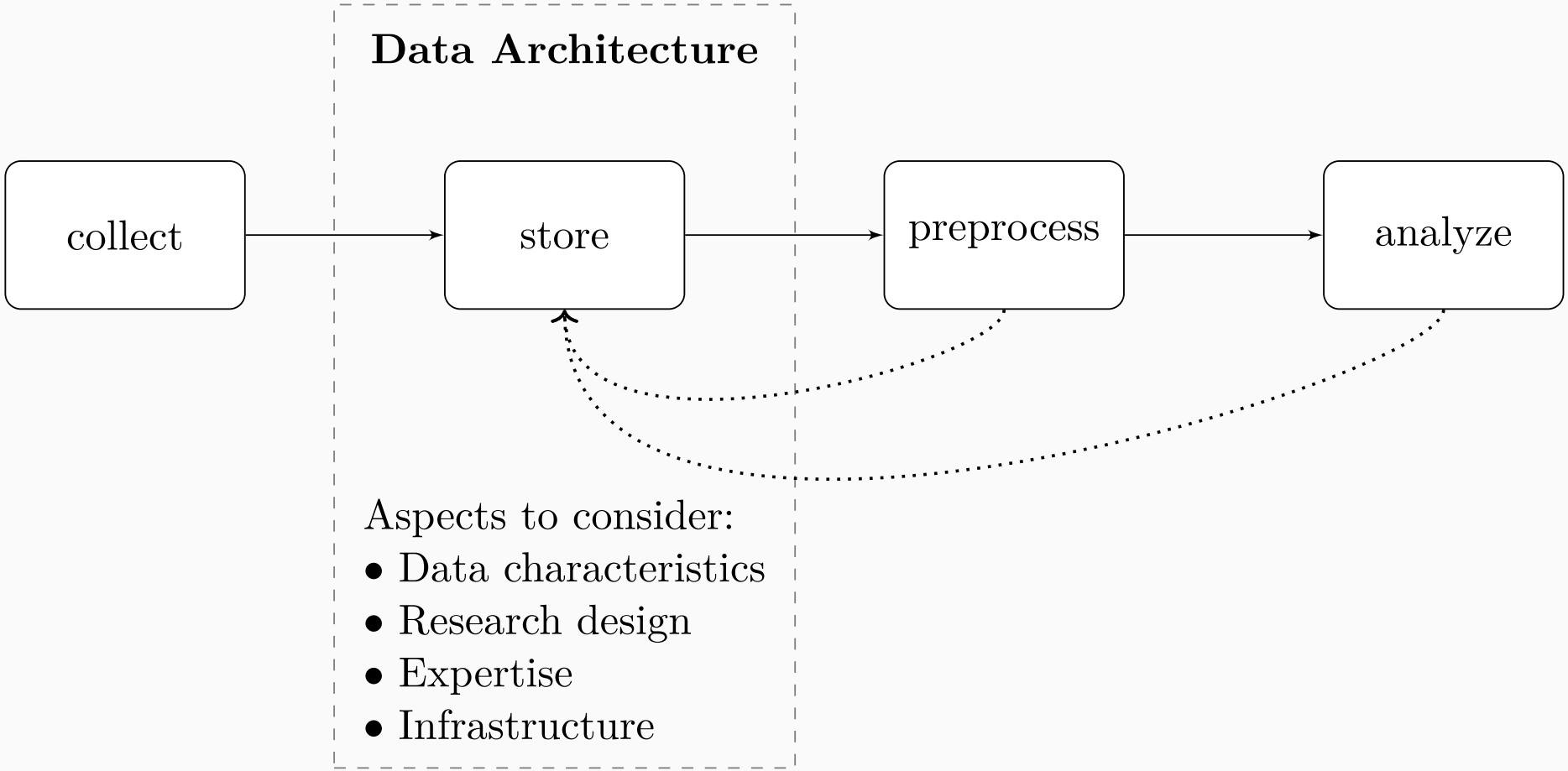
Storing data

Use of databases

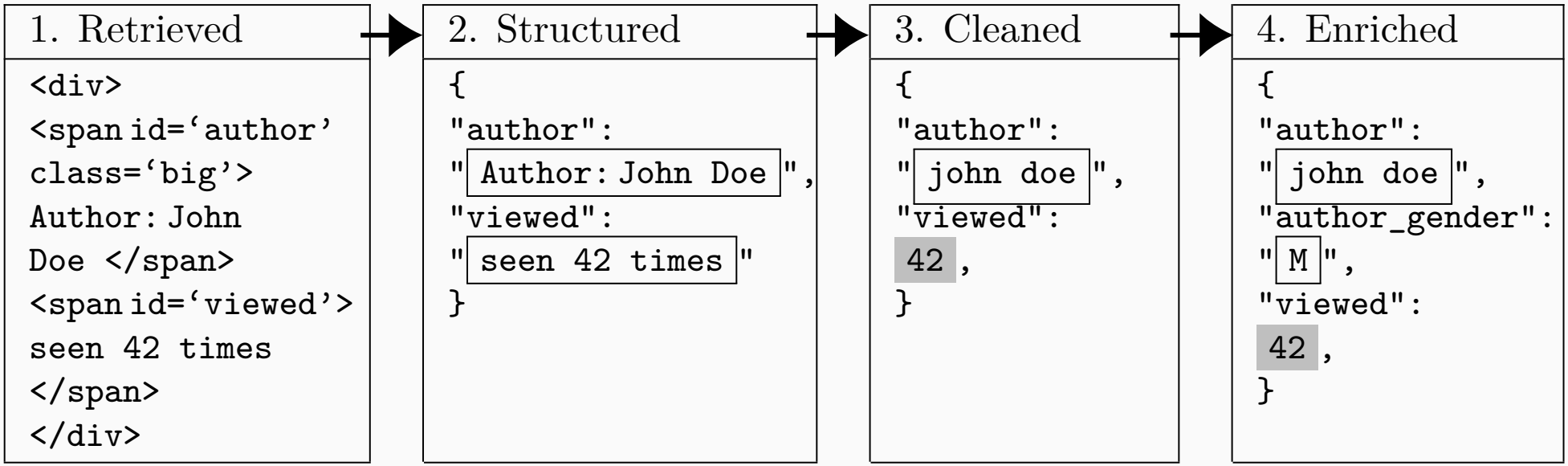
Storing data

- We can store our data in files (often, one CSV or JSON file)
- But that's not very efficient if we have large datasets; especially if we want to select subsets later on
- SQL-databases to store tables (e.g., MySQL)
- NoSQL-databases to store less structured data (e.g., JSON with unknown keys) (e.g., MongoDB, ElasticSearch)
- $\Rightarrow$ Günther, E., Trilling, D., & Van de Velde, R.N. (2018). But how do we store it? (Big) data architecture in the social-scientific research process. In: *Stuetzer, C.M., Welker, M., & Egger, M. (eds.): Computational Social Science in the Age of Big Data. Concepts, Methodologies, Tools, and Applications*. Cologne, Germany: Herbert von Halem.

Storing data



From retrieved data to enriched data



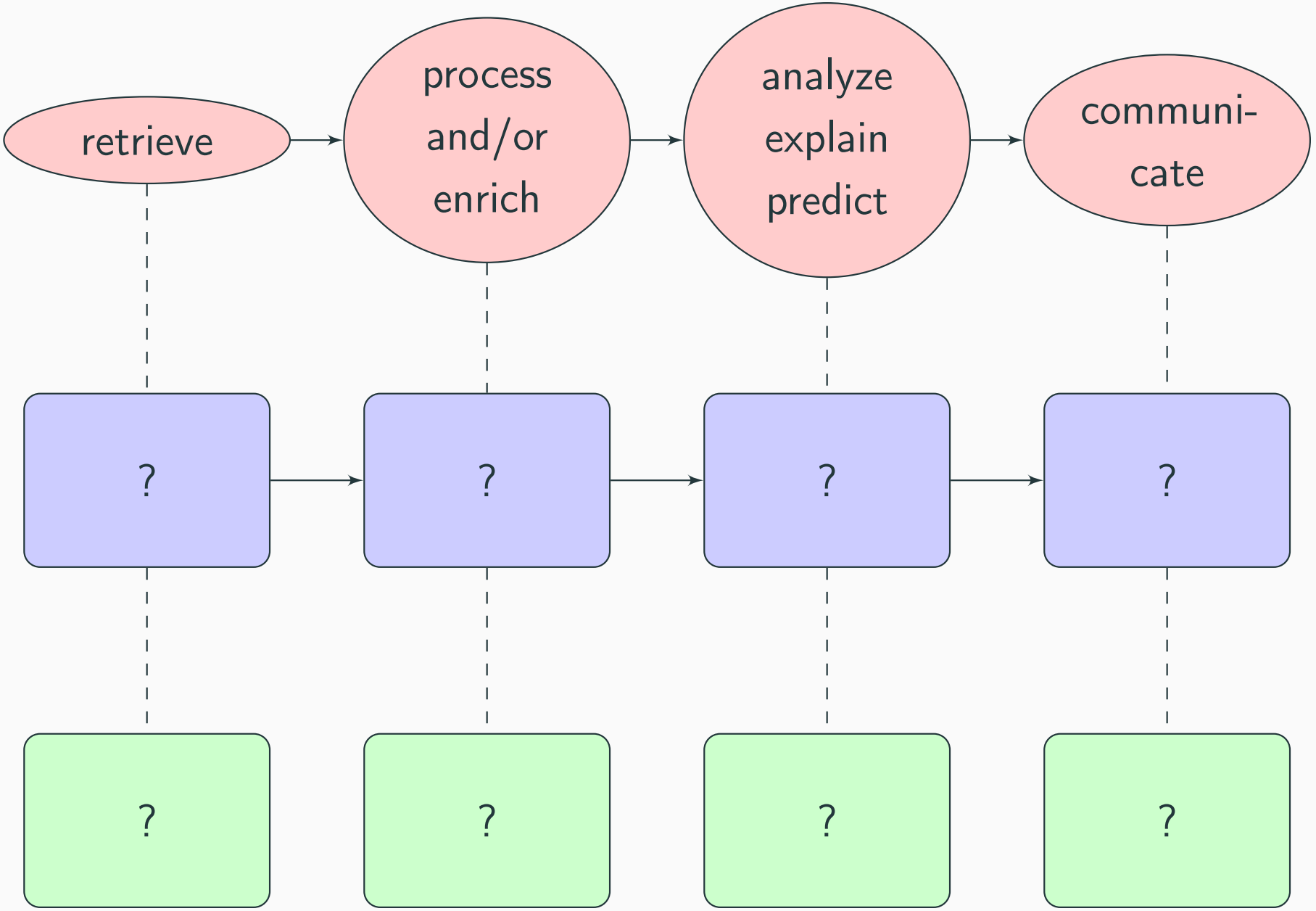
Looking forward

Looking forward

And now you...

Looking forward

Try to fill in the blanks for your personal CSS project


retrieve
process and/or enrich
analyze explain predict
communi- cate
?
?
?
?
?
?
?
?

Long story short:

Don't forget to plan the bigger picture

We will focus on machine learning this week. But for each technique we cover, think about how it fits in *your* workflow.

. . . and now lets get started!

Long story short:

Don't forget to plan the bigger picture

We will focus on machine learning this week. But for each technique we cover, think about how it fits in *your* workflow.

. . . and now lets get started!

The ACA toolkit

Types of Automated Content Analysis

The ACA toolkit

Top-down vs. bottom-up

	Methodological approach		
	Counting and Dictionary	*Supervised Machine Learning*	*Unsupervised Machine Learning*
Typical research interests and content features	visibility analysis sentiment analysis subjectivity analysis	frames topics gender bias	frames topics
Common statistical procedures	string comparisons counting	support vector machines naive Bayes	principal component analysis cluster analysis latent dirichlet allocation semantic network analysis

Some terminology

Supervised machine learning

You have a dataset with both predictor and outcome (independent and dependent variables; features and labels) — a *labeled* dataset. Think of regression: You measured x1, x2, x3 and you want to predict y, which you also measured

Unsupervised machine learning

You have no labels. (You did not measure y)

Again, you already know some techniques to find out how x1, x2,...x_i co-occur from other courses:

- Principal Component Analysis (PCA)
- Cluster analysis
- ...

Some terminology

Supervised machine learning

You have a dataset with both predictor and outcome (independent and dependent variables; features and labels) — a *labeled* dataset. Think of regression: You measured `x1`, `x2`, `x3` and you want to predict `y`, which you also measured

Unsupervised machine learning

You have no labels. (You did not measure `y`)

Again, you already know some techniques to find out how `x1`, `x2`,...,`x_i` co-occur from other courses:

- Principal Component Analysis (PCA)
- Cluster analysis
- ...

Some terminology

Supervised machine learning

You have a dataset with both predictor and outcome (independent and dependent variables; features and labels) — a *labeled* dataset. Think of regression: You measured `x1`, `x2`, `x3` and you want to predict `y`, which you also measured

Unsupervised machine learning

You have no labels. (You did not measure `y`)

Again, you already know some techniques to find out how `x1`, `x2`,...`x_i` co-occur from other courses:

- Principal Component Analysis (PCA)
- Cluster analysis
- ...

Some terminology

Supervised machine learning

You have a dataset with both predictor and outcome (independent and dependent variables; features and labels) — a *labeled* dataset. Think of regression: You measured `x1`, `x2`, `x3` and you want to predict `y`, which you also measured

Unsupervised machine learning

You have no labels. (You did not measure `y`)

Again, you already know some techniques to find out how `x1`, `x2`,...`x_i` co-occur from other courses:

- Principal Component Analysis (PCA)
- Cluster analysis
- ...

Some terminology

Supervised machine learning

You have a dataset with both predictor and outcome (independent and dependent variables; features and labels) — a *labeled* dataset. Think of regression: You measured `x1`, `x2`, `x3` and you want to predict `y`, which you also measured

Unsupervised machine learning

You have no labels. (You did not measure `y`)

Again, you already know some techniques to find out how `x1`, `x2`,...`x_i` co-occur from other courses:

- Principal Component Analysis (PCA)
- Cluster analysis
- ...

Final

This afternoon

Getting started

Getting started with the IMBD dataset

Backupslides basics

Backup slides in case we need to do more fundamentals

Datatypes

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's `"555"`).

But you can calculate `3 * int("5")`

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's "555").

But you can calculate `3 * int("5")`

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's `"555"`).

But you can calculate `3 * int("5")`

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']

list ages = [18,22,45]

dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']

list ages = [18,22,45]

dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice':  18, 'Bob':  22,
     'Cecile':  45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list  firstnames = ['Alice','Bob','Cecile']
      lastnames = ['Garcia','Lee','Miller']
list  ages = [18,22,45]
dict  agedict = {'Alice':  18, 'Bob':  22,
      'Cecile':  45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

Retrieving specific items

list `firstnames[0]` gives you the first entry

`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18

Python lingo

Retrieving specific items

list `firstnames[0]` gives you the first entry

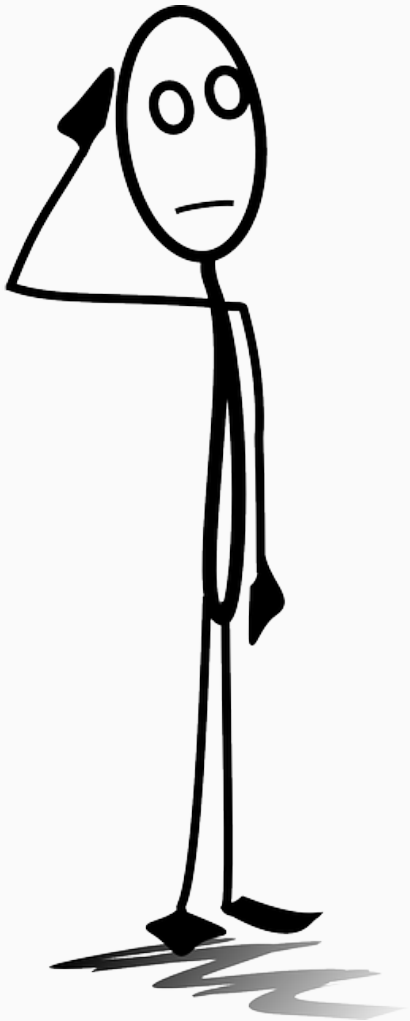
`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18



Think of at least two different ways of storing data about some fictious persons (first name, last name, age, phone number, . . .) using lists and/or dictionaries. What are the pros and cons?

Python lingo

Less frequent, but still useful datatypes

set	A collection in which each item is unique: `{1,2,3}`
tuple	Like a list, but *immutable*: `(1,2,2,2,3)`
defaultdict	A dict that does not raise an error but returns the "empty" value of its datatype (0 for int, `""` for str) if you try access a non-existing key (great for storing results and counting things!)
np.array	A list-like datatype provided by the `numpy` package optimized for efficient mathematical operations.
...	...

You will come across more later

Functions and methods

Python lingo

Functions

functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string `"scream"`

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

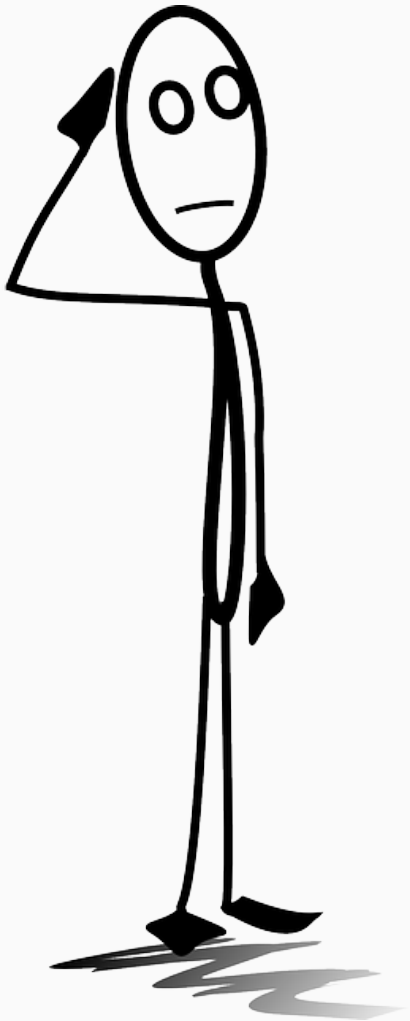
functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Some functions

```
1  len(x)       # returns the length of x
2  y = len(x)   # assign the value returned by len(x) to y
3  print(len(x)) # print the value returned by len(x)
4  print(y)     # print y
5  int(x)       # convert x to an integer
6  str(x)       # convert x to a string
7  sum(x)       # get the sum of x
```



How could you print the mean (average) of a list of integers using the functions on the previous slide?

Some methods

Some string methods

```
1 mystring = "Hi! How are you?"
2 mystring.lower() # return lowercased string (doesn't change original!)
3 mylowercasedstring = mystring.lower() # save to a new variable
4 mystring = mystring.lower() # or override the old one
5 mystring.upper() # uppercase
6 mystring.split() # Splits on spaces and returns a list ['Hi!', 'How', '
      are', 'you?']
```

We'll look into some list methods later.

⇒ **You can use TAB-completion in Jupyter to see all methods (and properties) of an object!**

Writing own functions

You can write an own function:

```
1  def addone(x):
2      y = x + 1
3      return y
```

Functions take some input ("argument") (in this example, we called it `x`) and *return* some result.

Thus, running

```
1  addone(5)
```

returns 6.

Writing own functions

Attention, R users! (maybe obvious for others?)

You *cannot** apply the function that we just created on a whole list – after all, it takes an int, not a list as input.

(wait a sec foruntil we cover for loops later today, but this is how you'd do it (by calling the function for each element in the list separately):):

```
1 mynumbers = [5, 3, 2, 4]
2 results = [addone(e) for e in mynumbers]
```

* Technically speaking, you could do this by wrapping the `map` function around your own function, but that's not considered "pythonic". Don't do it ;-)

Modifying lists & dicts

Modifying lists

Let's use one of our first methods! Each *list* has a method `.append()`:

Appending to a list

```
1 mijnlijst = ["element 1", "element 2"]
2 anotherone = "element 3" # note that this is a string, not a list!
3 mijnlijst.append(anotherone)
4 print(mijnlijst)
```

gives you:

```
1 ["element 1", "element 2", "element 3"]
```

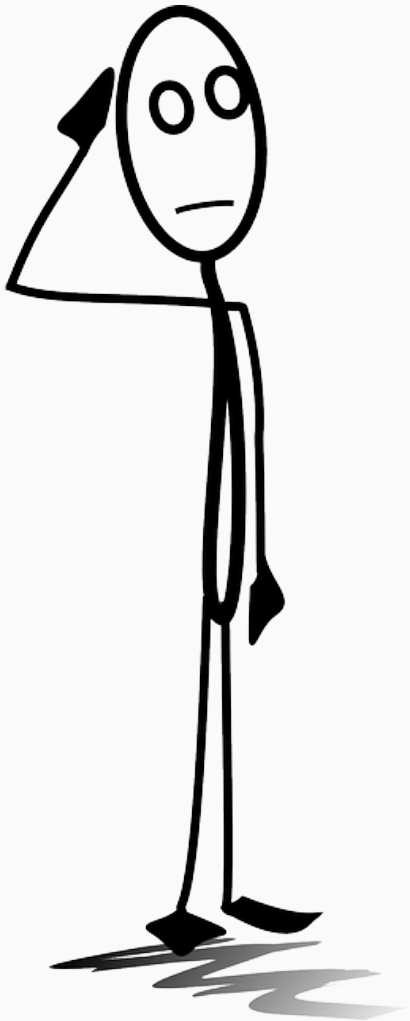
Modifying lists

Merging two lists (= extending)

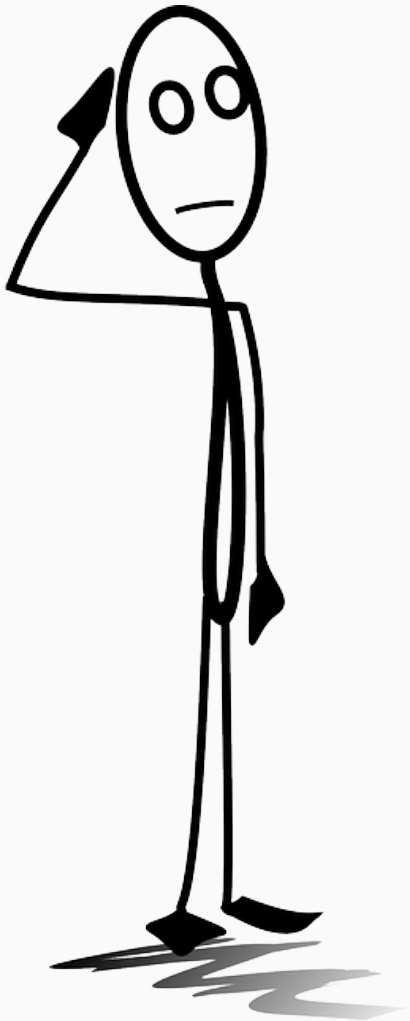
```
1 mijnlijst = ["element 1", "element 2"]
2 anotherone = ["element 3", "element 4"]
3 mijnlist.extend(anotherone)
4 print(mijnlijst)
```

gives you:

```
1 ["element 1", "element 2", "element 3", "element 4]
```



What would have happened if we had used `.append()` instead of `.extend()`?



Why do you think that the Python developers implemented `.append()` *and* `.extend()` *as methods of a list and not as functions?*

Modifying dicts

Adding a key to a dict (or changing the value of an existing key)

```
1 mydict = {"whatever": 42, "something": 11}
2 mydict["somethingelse"] = 76
3 print(mydict)
```

gives you:

```
1 {'whatever': 42, 'somethingelse': 76, 'something': 11}
```

If a key already exists, its value is simply replaced.

for, if/elif/else, try/except

How can we structure our program?

If we want to *repeat* a block of code, exectute a block of code only *under specific conditions*, or more generally want to structure our code, we use *indention*.

Indention: The Python way of structuring your program

- Your program is structured by TABs or SPACEs.
- Jupyter (or your IDE) handles (guesses) this for you, but make sure to not interfere and not to mix TABs or SPACEs!
- Default: four spaces per level of indention.

Indention

Structure

A first example of an indented block – in this case, we want to *repeat* this block:

```
1  agedict = {'Zeus': None, 'Denis': 96, 'Alice': 18, 'Rebecca': 20 , 'Bob
       ': 22, 'Cecile': 45}
2
3  myfriends = ['Alice','Bob','Cecile']
4
5  print ("The names and ages of my friends:")
6  for buddy in myfriends:
7          print (f"My friend {buddy} is {agedict[buddy]} years old")
```

Output:

```
1  My friend Alice is 18 years old
2  My friend Bob is 22 years old
3  My friend Cecile is 45 years old
```

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The for loop

1. Take the first element from `myfriends` and call it `buddy` (like `buddy = myfriends[0]`) (line 1)
2. Execute the indented block (line 2, but could be more lines)
3. Go back to line 1, take next element (like `buddy = myfriends[1]`)
4. Execture the indented block ...
5. ... repeat until no elements are left ...

The f-string (*formatted* string)

If you prepend a string with an `f`, you can use curly brackets texttt{} to insert the value of a variable

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (for or while loop) – once for each element from a list, or until a condition is reached (while statement)
- the block is only to be executed under specific conditions (if, elif, else statements)
- an alternative block should be executed if an exception is raised (try and except statements)
- a file is opened, but should be closed again when we're done with it (with statement)

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

Can we also loop over dicts?

Sure! But we need to indicate how exactly:

```
1  mydict = {"A":100, "B": 60, "C": 30}
2
3  for k in mydict: # or mydict.keys()
4      print(k)
5
6  for v in mydict.values():
7      print(v)
8
9  for k,v in mydict.items():
10     print(f"{k} has the value {v}")
```

Can we also loop over dicts?

The result:

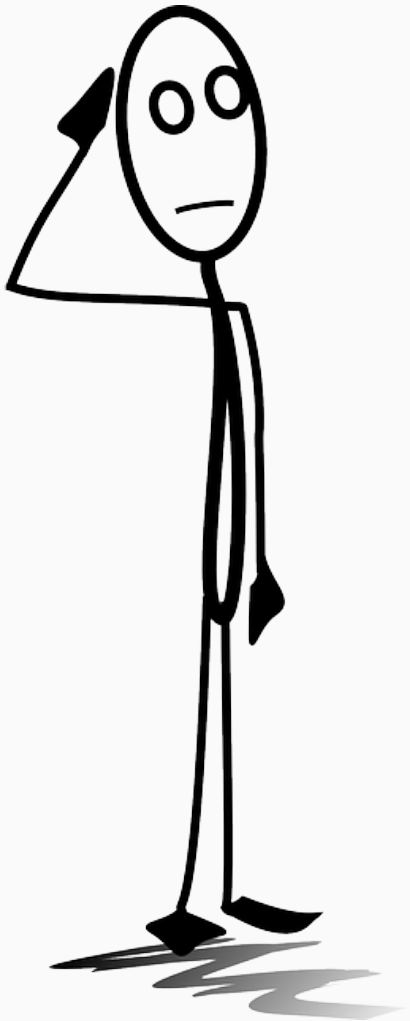
```
A
B
C

100
60
30

A has the value 100
B has the value 60
C has the value 30
```

if statements

Structure

Only execute block if condition is met

```
1  x = 5
2  if x <10:
3     print(f"{x} is smaller than 10")
4  elif x > 20:
5     print(f"{x} is greater than 20")
6  else:
7     print("No previous condition is met, therefore 10<={x}<=20")
```



Can you see how such an if statement could be particularly useful when nested in a for loop?

try/except

Structure

If executed block fails, run another block instead

```
1  x = "5"
2  try:
3     myint = int(x)
4  except:
5     myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
1  mylist = ["5", 3, "whatever", 2.2]
2  myresults = []
3  for x in mylist:
4      try:
5          myresults.append(int(x))
6      except:
7          myresults.append(None)
8  print(myresults)
```

try/except

Structure

If executed block fails, run another block instead

```
x = "5"
try:
   myint = int(x)
except:
   myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
mylist = ["5", 3, "whatever", 2.2]
myresults = []
for x in mylist:
    try:
        myresults.append(int(x))
    except:
        myresults.append(None)
print(myresults)
```

Bonus: Python goodies

List comprehensions

Structure

A for loop that `.append()`s to an empty list can be replaced by a one-liner:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = []
3 for x in mynumbers:
4     mysquarednumbers.append(x**2))
```

is equivalent to:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

Structure

A for loop that `.append()`s to an empty list can be replaced by a one-liner:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = []
3 for x in mynumbers:
4     mysquarednumbers.append(x**2))
```

is equivalent to:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

A very pythonic construct

- Every for loop can also be written as a for loop that appends to a new list to collect the results.
- For very complex operations (e.g., nested for loops), it can be easier to write out the full loops.
- But mostly, list comprehensions are really great! (and much more concise!)

$\Rightarrow$ **You really should learn this!**

Generators

Structure

A lazy for loop (or function) that only generates its next element when it is needed:

You can create a generator just like a list comprehension (but with `()` instead of `[]`):

```
1 mynumbers = [2,1,6,5]
2 squaregen = (x**2 for x in mynumbers) # these are NOT calculated yet
3 for e in squaregen:
4     print(e)           # only here, we are calculating the NEXT item
```

Or like a function (but with `yield` instead of `return`):

```
1 def squaregen(listofnumbers):
2     for x in listofnumbers:
3         yield(x**2)
4 mygen = squaregen(mynumbers)
5 for e in mygen:
6     print(e)
```

Generators

Structure

A lazy for loop (or function) that only generates its next element when it is needed:

You can create a generator just like a list comprehension (but with `()` instead of `[]`):

```
1 mynumbers = [2,1,6,5]
2 squaregen = (x**2 for x in mynumbers) # these are NOT calculated yet
3 for e in squaregen:
4     print(e)           # only here, we are calculating the NEXT item
```

Or like a function (but with `yield` instead of `return`):

```
1 def squaregen(listofnumbers):
2     for x in listofnumbers:
3         yield(x**2)
4 mygen = squaregen(mynumbers)
5 for e in mygen:
6     print(e)
```

Generators

A very memory and time efficient construct

- Every function that *returns* a list can also be written as a generator that *yields* the elements of the list
- Especially useful if
  - it takes a long time to calculate the list
  - the list is very large and uses a lot of memory (hi big data!)
  - the elements in the list are fetched from a slow source (a file, a network connection)
  - you don't know whether you actually will need all elements

$\Rightarrow$ **You probably don't need this right now, but (a) it will come in very handy once you deal with web scraping or very large collections, and (b) you may come across generators in some examples**

Make sure you understood all of today's concepts.

Also read Chapter 4 and ask questions if needed. If you want do some exercises with basic python, please see here: https://github.com/annekroon/gesis-machine-learning/blob/main/day1/exercises-basicpython/exercises.md