Big Data and Automated Content Analysis
Week 2: »Programming concepts and writing code«
Wednesday

Anne Kroon
a.c.kroon@uva.nl, @annekroon

April 11, 2022

UvA RM Communication Science

Today

Datatypes

Functions and methods

Modifying lists and dictionaries

for, if/elif/else, try/except

Bonus: Python goodies

Next steps

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

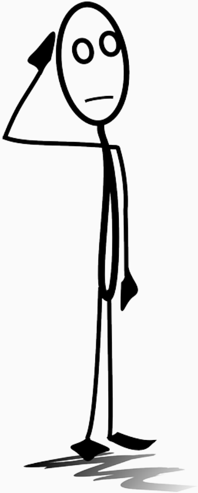
`firstnames[-2]` gives you the one-but-last entry

`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

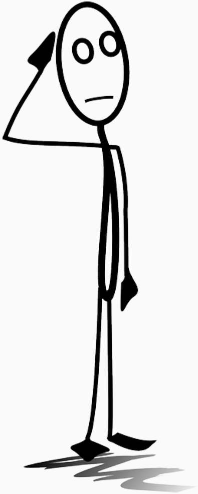
functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

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

```
len(x)          # returns the length of x
y = len(x)      # assign the value returned by len(x) to y
print(len(x))   # print the value returned by len(x)
print(y)        # print y
int(x)          # convert x to an integer
str(x)          # convert x to a string
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

```
mystring = "Hi! How are you?"
mystring.lower()   # return lowercased string (doesn't change
↪  original!)
mylowercasedstring =  mystring.lower()  # save to a new variable
mystring =  mystring.lower()  # or override the old one
mystring.upper()   # uppercase
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:

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

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

Thus, running

```
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):):

```
mynumbers = [5, 3, 2, 4]
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

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

gives you:

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

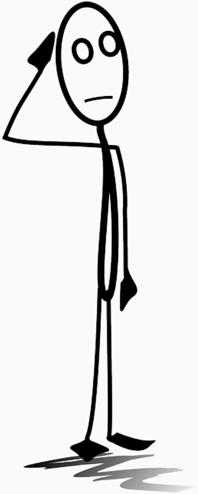
Modifying lists

Merging two lists (= extending)

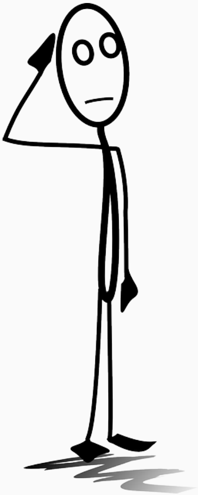
```
mijnlijst = ["element 1", "element 2"]
anotherone = ["element 3", "element 4"]
mijnlist.extend(anotherone)
print(mijnlijst)
```

gives you:

```
["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)

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

gives you:

```
{'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:

```
agedict = {'Zeus': None, 'Denis': 96, 'Alice': 18, 'Rebecca': 20 ,
↪  'Bob': 22, 'Cecile': 45}

myfriends = ['Alice','Bob','Cecile']

print ("The names and ages of my friends:")
for buddy in myfriends:
        print (f"My friend {buddy} is {agedict[buddy]} years old")
```

Output:

```
My friend Alice is 18 years old
My friend Bob is 22 years old
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 `{}` to

What happened here?

```python
for buddy in myfriends:
    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 `:` and indention indicate 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?

```python
for buddy in myfriends:
    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 `:` and indention indicate 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?

```python
for buddy in myfriends:
    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 `:` and indention indicate 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?

```
for buddy in myfriends:
    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 `:` and indention indicate 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?

```
for buddy in myfriends:
    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 : and indention indicate 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?

```
for buddy in myfriends:
    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 `:` and indention indicate 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:

```
mydict = {"A":100, "B": 60, "C": 30}

for k in mydict:    # or mydict.keys()
    print(k)

for v in mydict.values():
    print(v)

for k,v in mydict.items():
    print(f"{k} has the value {v}")
```

Can we also loop over dicts?

The result:

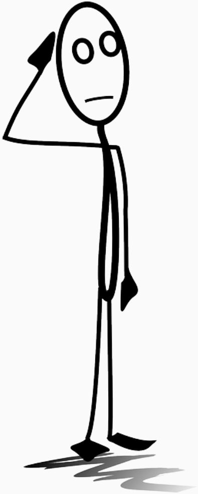
```
A
B
C

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

```python
x = 5
if x <10:
   print(f"{x} is smaller than 10")
elif x > 20:
   print(f"{x} is greater than 20")
else:
   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

```
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)
```

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

List comprehensions

Structure

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

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

is equivalent to:

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

Optionally, we can have a condition:

```
mynumbers = [2,1,6,5]
```

List comprehensions

Structure

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

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

is equivalent to:

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

Optionally, we can have a condition:


```
mynumbers = [2,1,6,5]
```

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 `[]`):

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

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

```
def squaregen(listofnumbers):
    for x in listofnumbers:
        yield(x**2)
mygen = squaregen(mynumbers)
for e in mygen:
    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 []):

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

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

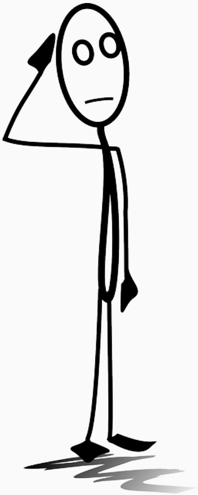
```
def squaregen(listofnumbers):
    for x in listofnumbers:
        yield(x**2)
mygen = squaregen(mynumbers)
for e in mygen:
    print(e)
```

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

⇒ 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



Any questions?

Next steps

Make sure you understood all of today's concepts.

Re-read Chapter 3 and read Chapter 4. Ask questions on Thursday if needed.

I prepared exercises to work on *during* the Thursday meeting (alone or in teams):
https://github.com/uvacw/teaching-bdaca/blob/main/6ec-course/week02/exercises.md