RICE SOCIAL SCIENCES
Department of Linguistics

control flow & dictionaries

LING 430 Computational Linguistics Fall 2026
lecture 2b
Sep 3 2026

Siyu Liang
siyu.liang@rice.edu

last time

- types, string, and lists
- indexing and slicing

agenda

- booleans
- if statement
- for statement
- dictionaries
- tuples & sets

today's notebook

class02b.ipynb

- course website → week 2 → activity
- **File → Save a copy in Drive**

control flow

- so far, our code runs one line after the next

```
phrase = "a " + word
phrase = "an " + word
```

- but there are decisions we need to make depending on the data
  - add `"an"` only when `word` begins with a vowel
- control flow: choosing which lines to run, and how many times

booleans & if

boolean

- a *boolean expression* asks a question; the answer is `True` or `False`

  `"rose" == "Rose"`

  → False

  `"rose" in ["Rose", "is", "a", "rose"]`

  → True

  `len("linguistics") > 5`

  → True

boolean

and, or, not

- `and` needs **both** to be True
  `len("rose") > 3 and "z" in "rose"`
  → False
- `or` needs **either** to be True
  `"rose".endswith("e") or "rose".endswith("s")`
  → True
- `not` **flips** the boolean value
  `not "rose" == "Rose"`
  → True

if

- `if` runs a block of code only when a condition is true

```
word = "singing"
if word.endswith("ing"):
    print(word, "may be a gerund or participle")
```

→ singing may be a gerund or participle

- **`if`**: keyword (a decision is coming)
- **`word.endswith("ing")`**: the condition (anything True or False)
- **`:`** “a block of code is coming”
- **indented lines**: run only if the condition holds

if

- indentation matters!
  - indentation is the Python way of bracketing in conditionals
  - **four** spaces
  - Colab automatically indents for you after `:` and return
  - Colab also silently converts one tab to four spaces

elif & else

- conditional statements can be organized top-down
- run until the **first** true statement, and skip the rest

```
word = "she"
if word in ["the", "a", "an"]:
    print("article")
elif word in ["she", "he", "they", "it"]:
    print("pronoun")
else:
    print("something else, most of the lexicon!")
```

→ pronoun

elif & else

- statements could always get skipped if in incorrect order

```
word = "glass"
if word.endswith("s"):
    print("looks plural")
elif word.endswith("ss"):
    print("no it doesn't")
```

  → looks plural
- the `ss` branch can **never** validate
  - everything ending in ss also ends in s
- rule: specific before general

elif & else

elif vs separate ifs

- two separate ifs: both are checked

```
if word.endswith("s"):
    print("ends in s")
if word.endswith("ss"):
    print("ends in ss")
```

→ ends in s
→ ends in ss

- separate `if`s each get their turn; an `elif` chain stops at the first match
- how to choose
  - are these *alternatives*, or independent checks?

if

a linguistic rule: a or an

- English article **allomorphy**

```
word = "owl"
if word[0] in "aeiou":
    print("an", word)
else:
    print("a", word)
```

→ an owl

if

a linguistic rule: a or an

- how about these words
  `word = "hour"`
  → a hour
  `word = "university"`
  → an university
  `word = "hermès bag"`
  → ???
- the real rule is phonological
- our condition sees only spelling
- but maybe we can try to handle these special cases

nested if

- an if can sit inside another if
- day → days, fly → flies, city → cities

```
noun = "day"
if noun.endswith("y"):
    if noun[-2] in "aeiou":
        print(noun + "s")
    else:
        print(noun[:-1] + "ies")
else:
    print(noun + "s")
```

→ days

- the inner `if` only runs when the outer condition held

nested if

nested if vs and

- you can also use `and`

```
if noun.endswith("y") and noun[-2] not in "aeiou":
    print(noun[:-1] + "ies")
else:
    print(noun + "s")
```

→ days

activity

English a/an allomorphy expanded

- see the Python notebook for a starter code
- *hour*, *heir*: “an”
- *university*, *unicorn*, *uniform*: “a”
- add a nested `if` inside each branch to handle these
- change `word` and run again to test each case

for

for

- `for` runs a block of code once for each item in a list

```
words = ["colorless", "green", "ideas", "sleep", "furiously"]
for w in words:
    print(w, len(w))
```

→ colorless 9
→ green 5
→ ideas 5
…

- **`for`**: keyword (a repetition is coming)
- **`w`**: **your name** for each item in turn
- **`in words`**: the list to walk through
- **`:` + indent**: the body of code that runs once per item

for

string loop

- a list loops item by item; a string loops **character by character**

```
for ch in "cat":
    print(ch)
```

→ c
→ a
→ t

while & range

- `for i in range(3):`
  - the numbers 0, 1, 2; looping a fixed number of times
- `while condition:`
  - repeat until something changes
- working with language data will include a lot of `for`
  - we have the words/sentences/paragraphs, we walk them

while & range

- `while` repeats until the condition fails

```
word = "rererun"
while word.startswith("re"):
    word = word[2:]
word
```

→ 'run'

- a `for` loop couldn’t do this: we don’t know in advance how many layers there are

for

counting occurrences

- count how many words match a pattern

```
adverb_count = 0
for w in words:
    if w.endswith("ly"):
        adverb_count = adverb_count + 1
adverb_count
```

→ 1

for

keeping track of items

- keep track of items that satisfy a certain criteria

```
adverbs = []
for w in words:
    if w.endswith("ly"):
        adverbs.append(w)
adverbs
```

→ ['furiously']

- same loop, same condition, with an empty list and `.append()` instead of a counter

for

looping over structured data

- each item is itself a list, so index into it inside the body

```
glosses = [["inu", "dog"], ["neko", "cat"], ["tori", "bird"]]
for pair in glosses:
    print(pair[0], "=", pair[1])
```

→ inu = dog
→ neko = cat
→ tori = bird

activity

- use while to convert ‘great-great-great-grandmother’ to ‘grandmother’
- and then do that for a list of relatives

dictionaries

dictionary

- a *dictionary* is a collection of **key : value** pairs
  ```
  myDict = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
  myDict['quantity']
  ```
  → 4
- lookup is by **key**, not by position
  - efficient, like a real dictionary's headwords
- **mutable**, like lists
  - pairs can be added, changed, and removed

dictionary

operations

- adding, checking, and looking up safely
  ```
  gloss = {"inu": "dog", "neko": "cat"}
  gloss["tori"] = "bird"       # add a pair
  "inu" in gloss
  ```
  → True
  ```
  gloss["uma"]
  ```
  → KeyError: 'uma'
  ```
  gloss.get("uma", "???")
  ```
  → '???'
- `in` checks the keys; a missing key raises `KeyError`
- `.get(key, default)` asks politely, and hands back a fallback

dictionary

counting words

- count every word in a text at once

```
words = "a rose is a rose is a rose".split()
counts = {}
for w in words:
    counts[w] = counts.get(w, 0) + 1
counts
```

→ {'a': 3, 'rose': 3, 'is': 2}

- `.get(w, 0)`: current count if seen, else start at 0, so no `KeyError`

dictionary

recap

operation	example	result
create	`gloss = {"inu": "dog"}`	
look up	`gloss["inu"]`	`'dog'`
add / change	`gloss["tori"] = "bird"`	
membership	`"inu" in gloss`	`True`
safe lookup	`gloss.get("uma", 0)`	`0`
size	`len(gloss)`	`2`
remove	`del gloss["inu"]`	

dictionary

looping over one

- a `for` loop over a dictionary walks its **keys**

```
for w in counts:
    print(w, counts[w])
```

→ a 3
→ rose 3
→ is 2

- look the value up inside the body

dictionary

values can be anything

- a value can be a list, or another dictionary
  ```
  gloss = {"duck": ["Ente", "pato"], "goose": ["Gans", "ganso"]}
  gloss["duck"][1]
  ```
  → 'pato'
- read the brackets left to right
  - look up the key, then index the value

tuple & set

- *tuples* are like lists but **immutable** and **ordered**
  - `("inu", "dog")`
- *sets* are **mutable**, **unordered** collections of unique items

  `set(words)`

  → {'a', 'is', 'rose'}

types

recap

type	holds	ordered?	mutable?	used for
str	characters	yes	no	texts
list	anything	yes	yes	tokenized texts
dict	key : value pairs	by key	yes	counts, lexicons
tuple	anything	yes	no	fixed pairs
set	unique items	no	yes	vocabularies

Turkish vowel harmony

- Turkish has eight vowels in two groups governed by vowel harmony
  - **back**: a ı o u
  - **front**: e i ö ü
- a suffix vowel agrees with backness of the word's **last** vowel
  - plural marking, lAr, can be either *-lar* or *-ler*
    - *kebap* → *kebaplar*
    - *ev* → *evler*
    - *okul* → *okullar*
    - *otobüs* → *otobüsler*

the locative

- **locative** is a case that indicates location, meaning “in, on, at, …”
- In Turkish, locatives are expressed with the suffix *–DA*
  - *ev* → *evde*, *okul* → *okulda*
- which agrees with the **voicing** of the final consonant
  - after a **voiceless** consonant (ç f h k p s ş t), *d* becomes *t*
  - *kebap* → *kebapta*, *yoğurt* → *yoğurtta*, *otobüs* → *otobüste*
- in summary
  - four forms: *-da -de -ta -te*

activity

the Turkish locative

- given a list of words, print the locative form of every word
- tips
  - find the last vowel of the word
    - back vowel → *a*, front vowel → *e*
  - find the last consonant
    - last letter voiceless → *t*, otherwise *d*
- extra challenge
  - add the plural first, then the locative
    - *ev* → *evlerde* “in the houses”, *otobüs* → *otobüslerde*

next time

- Tuesday
  - language data & nltk
  - NLTK Book, Ch. 1, §§1 and 3: Computing with Language
- **HW 1 due next Thursday, Sep 10**