Meeting 2 – data structures and data abstraction

Dan

Recap: Higher-order functions

- Example of function that takes a function as input: functions with the same behavior can be refactored into a single function that takes a function. The next example is similar to the sum notation from mathematics: $\sum_{i=0}^{n} f(i)$.

```
def sum_numbers(n):
    total = 0
    while n > 0:
        total += n
        n -= 1
    return total

def sum_squares(n):
    total = 0
    while n > 0:
        total += square(n)
        n -= 1
    return total

def sum_cubes(n):
    total = 0
    while n > 0:
        total += cube(n)
        n -= 1
    return total
```

A more modular way of implementing the above code is by making use of higher-order functions:

```
def sum_sequence(n, func):
    total = 0
    while n > 0:
        total += func(n)
        n -= 1
    return total
```

Q Define `sum_numbers`, `sum_squares` and `sum_cubes` in terms of `sum_sequence`.

- Example of function that takes a function and returns a function. In mathematics, the derivative can be seen as an example of a function that takes a function and returns a function. Taking an example, if $f(x) = x^2$, then $\frac{df}{dx} = 2x$ which is again a function. We can say that the derivative is a higher-order function, which for each function from $\Re \to \Re$ gives us another function from $\Re \to \Re$, that is, $\frac{d}{dx} : (\Re \to \Re) \to (\Re \to \Re)$. Or in programming terms:

```
def derivative(f):
    def f_tag(x):
        x1, x2 = x - 0.0001, x + 0.0001
        y1, y2 = f(x1), f(x2)
        return (y2 - y1) / (x2 - x1)
    return f_tag
```

Further reading:

- Decorators in Python – special syntax that allows to modify the behaviour of a function based on another function.

Recap: Homework

- Evaluation of function arguments (for the `what_if` exercise)
- Multiple arguments to functions using `*args`
- Multiple ways of defining a string: single, double, triple quotes

Overview

1. Tour of Python data structures: lists, dictionaries, strings, tuples
2. Data abstraction

Sequences

- Ordered collection of values

Lists

- The `len` function
- Indexing and slicing: *e.g.*, `xs[3]`, `xs[1:-2]`
- Indexing starts from 0
- The `in` operator – testing membership
- The `*` operator

- Empty list `[]` is falsy
- Exercises: Lab 04 Q1

Sequence iteration

- Sequence iteration, or `for` loops
- Ranges: using `for _ in range(n)` to iterate instead of the while loop
- Exercises: Lab 04 Q3 (Fall 2017)

Sequence processing

- List comprehensions – similar to the mathematical notation $\{x^3 : x \in [0, \ldots, n]\}$; in Python that would be `[i ** 3 for i in range(n)]`.
- General expression: `[<exp> for <name> in <sequence> if <predicate>]`
- Common operations on lists: *(i)* mapping: `[a] → [a]`; *(ii)* filtering: `[a] → [a]`; *(iii)* aggregation: `[a] → a` examples include built-in functions like `sum`, `min`, `max`. Mapping and filtering can be achieved using list comprehensions

Strings

- Single and double quotes
- Multi-line literals – using triple quotation marks
- Operators: `in`, `+`, `\*`

Dictionaries

- Mapping correspondence relationships: key-value pairs
- Examples: roman numerals, month to id
- Indexing: `dict[key]`
- Methods: `keys`, `values`, `items`, `get` (can a default value if key is missing)
- Constructing dictionaries from list of pairs
- Restrictions: *(i)* keys are immutable; *(ii)* only a single value for a given key
- Dictionary comprehensions: `{v: k for k, v in d.items()}`
- Common uses of dictionaries: invert a dictionary, count words using a dictionary and the defaultdict approach
- Exercises: Lab 05 Q1 (Fall 2017)

Data abstraction

- Data is usually compound: date (year, month, date), complex numbers (real and imaginary part), geographical position (latitude and longitude)

- It is easier to operate with data as a whole.
- The idea of data abstraction is to pack compound data into an abstract representation using *constructors* and allow to access parts using *selectors*
- Idea: We can use the abstraction irespective of how it's implemented!
- Probably you are familiar from OOP with this idea, but for the time being we will use built-in data structures to implement the compound data and functions as constructors and selectors
- Example: city data abstraction (Lab 04)
- Abstraction layers:

```
cities as a whole
---- distance, closer city ----
cities as name, latitude and longitude
--- make_city, get_name, get_lat, get_lon ----
cities as tuples    | cities as dicitionaries
---- tuple, getitem | dict, getitem ----
```

- The higher layers are agnostic to the representation of the cities – this makes the code more modular