Lambda

Erinnerung: Imperativ vs. Deklarativ

- Imperative Programmierung:
  - Programm besteht aus einer Abfolge von Befehlen
- Deklarative Programmierung:
  - Programm besteht aus Beschreibung des Ergebnisses
  - Funktionale Programmierung:
    z.B. $f(x) = x^2$
  - Grundsätze:
    - Code besteht aus Funktionen, welche für Eingabe die Ausgabe beschreiben
    - Abstraktion!!!
    - Keine *Nebeneffekte*
- Python ist eher imperativ aber besitzt auch funktionale Komponenten

Lambda-Kalkül λ

- Formale Sprache zur Beschreibung von Funktionen
- Sprachelemente besitzen keinen Zustand, lediglich Ein- und Ausgabe
- Gegenentwurf zur Turingmaschine
  - Welche übrigens die Basis imperativer Sprachen ist
- https://de.wikipedia.org/wiki/Lambda-Kalk%C3%BCl
- https://www.youtube.com/watch?v=eis11j_iGMs

Lambda in Python

- Idee: Definition von *anonymen Funktionen*
  - Anonyme Funktion = Funktion ohne Namen
  - Können komplex sein aber meist eher kleine Funktionen
- Syntax:

```
lambda x, y: x + y
```

- Keyword zum Einleiten einer Lambda Funktion
- Parameterliste -> Eingabe
- Ausdruck („Funktionskörper“) -> Verarbeitung & Ausgabe
- Fun Fact: GvR wollte kein Lambda in Python haben, konnte sich aber nicht durchsetzen

Beispiel: Identitätsfunktion

- Funktion, die lediglich den Parameter zurückgibt (-> identische Abbildung)

```
3  def identity_f(x):
4      return x
5
6  identity_l = lambda x: x
7
8  for func in [identity_f, identity_l]:
9      print(func)
10     print(type(func))
11     print(func(10), "\n")
```

```
<function identity_f at 0x0000022AB3C257E0>
<class 'function'>
10

<function <lambda> at 0x0000022AB3C25900>
<class 'function'>
10
```

- IIFE („iffy“)

Immediately Invoked Function Expression

```
# ohne Variablenzuweisung:
print((lambda x: x)(10))
```

Lambda in Python

- Mit Vorsicht genießen:
- Hohe Abstraktion -> geringere Lesbarkeit
- Viele Entwickler verstehen Lambda-Funktionen nicht
- Lambda widerspricht dem imperativen Programmierparadigma
- Mischen von Paradigmen ist ok aber kann verwirren
- Alternative (nicht empfohlen!): One line Funktionsdefinition
  - Z.b. `def f(x, y): return x + y` statt `lambda x, y: x + y`

Weitere Besonderheiten

- Lambda kann keine Befehle beinhalten (z.b. raise)
- Keine Type Hintings (z.b. lambda x:int: x + 1 -> SyntaxError)
- Das lässt sich alles in Lambda-Funktionen weiterhin verwenden:
  - Keyword arguments
  - *args
  - **kwargs
- Für Fallunterscheidungen:

```
lambda x: <true_case> if <condition> else <false_case>
```

z.b. `(lambda x: "gerade" if x%2 == 0 else "ungerade")`

Wann nutzt man lambda?

- Beliebter Use Case: In Kombination mit **map()** und **filter()**
  - `map(func, iterable)`
    # Erzeugt einen Iterator, der die Funktion func auf jedes Element von iterable anwendet

```
17  nrs = [1, 2, 3, 4, 5]
18  double = list(map(lambda x: x*2, nrs))
19  triple = [i for i in map(lambda x: x*3, nrs)]
20  print(double, triple)
```

```
[2, 4, 6, 8, 10] [3, 6, 9, 12, 15]
```

  - filter(func, iterable)
    # Erzeugt einen Iterator, der alle Elemente von iterable zurückgibt, bei denen func True ist

```
22  nrs = [1, 2, 3, 4, 5]
23  odd_nrs = list(filter(lambda x: x % 2 == 1, nrs))
24  even_nrs = list(filter(lambda x: x % 2 == 0, nrs))
25  print(odd_nrs, even_nrs)
26
```

```
[1, 3, 5] [2, 4]
```

Wann nutzt man lambda?

- Für die Sortierfunktionen (bzw. alle Funktionen, mit Parameter key):
- Z.b. Dictionary anhand von values sortieren:

```
28  dicc = {"a": 1, "b": 3, "c": 2}
29  print(dict(sorted(
30      dicc.items(),
31      key=lambda x: x[1]
32  )))
```

```
{'a': 1, 'c': 2, 'b': 3}
```

Übung & Hausi

- https://realpython.com/quizzes/python-lambda/
- https://www.w3resource.com/python-exercises/lambda/index.php