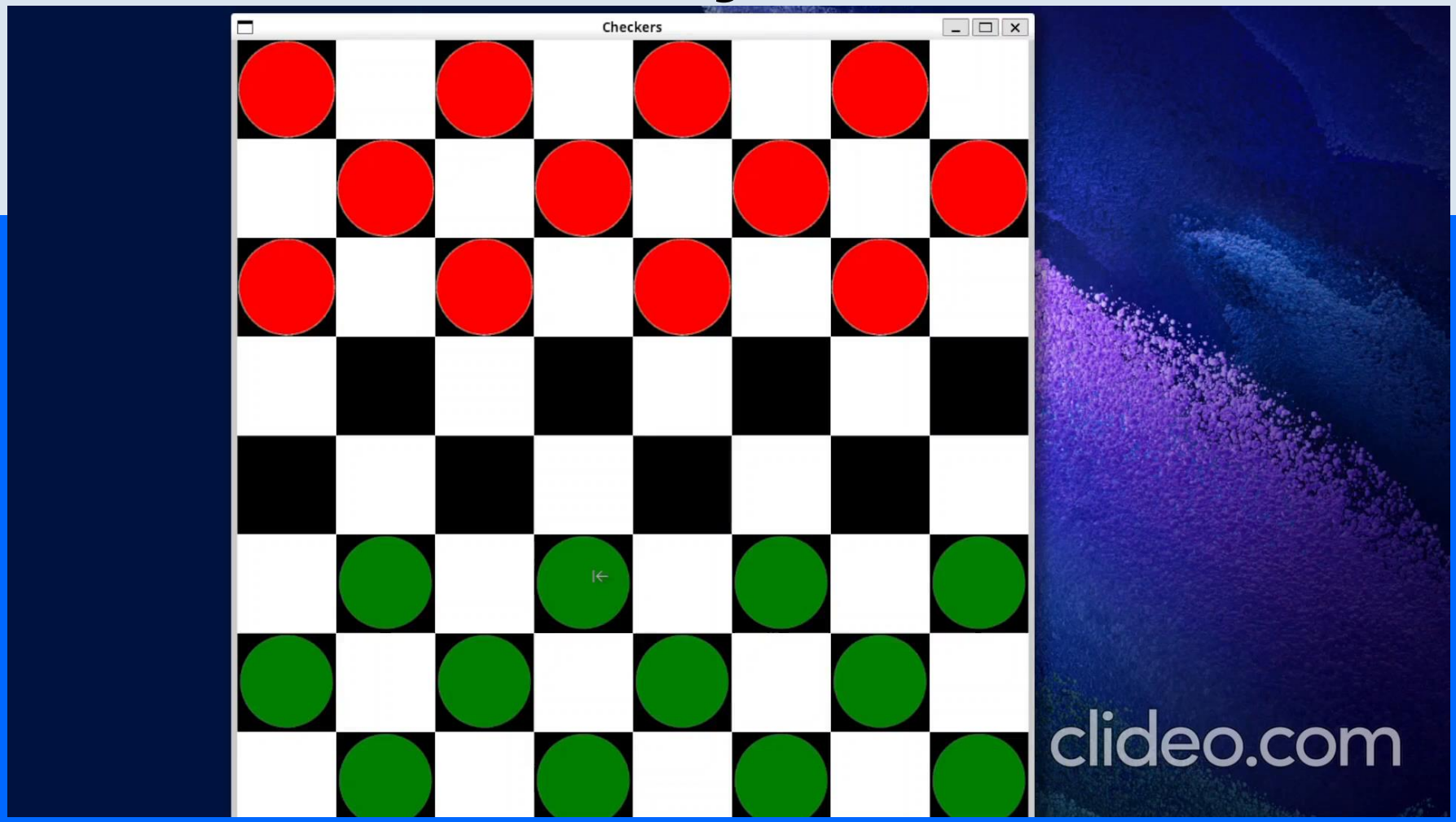
Funktionale Programmierung

Vergleich Go und Python

Projekt

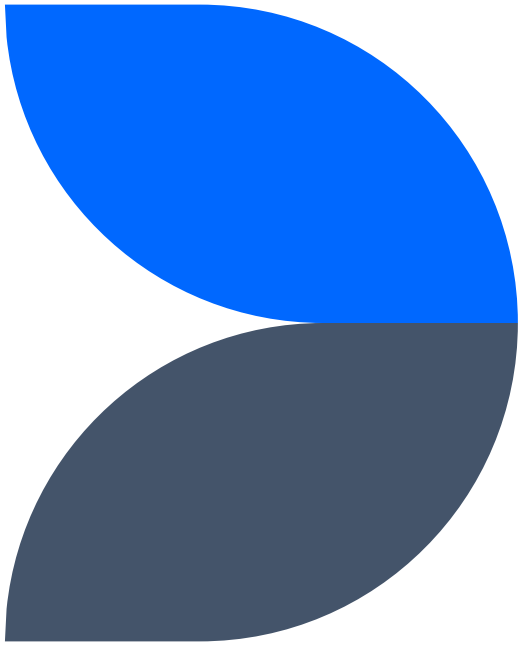


Minimax Algorithmus zum Lösen von Dame

Agenda

- Python versus Go
- Higher-Order Functions
- Pure Functions
- Immutability
- Recursion
- Lazy Evaluation

Python versus Go



Python versus Go

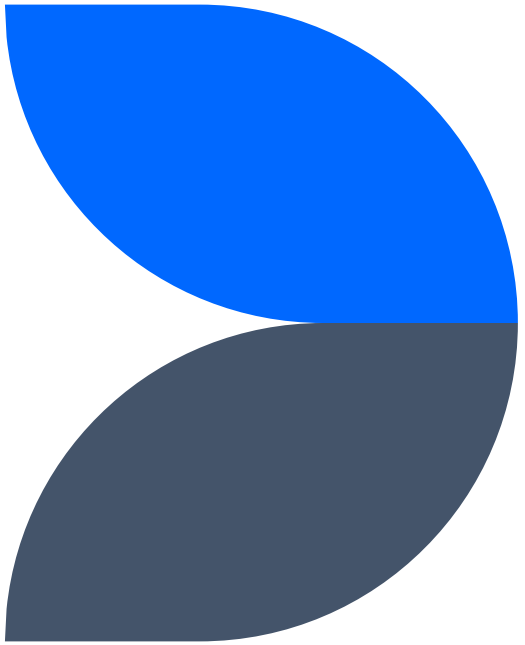
Go

- Statisch typisiert
- Kompiliert
- Multiparadigma

Python

- Dynamisch typisiert
- Interpretiert
- Multiparadigma

Higher Order Functions



Higher Order Functions

Go

- Behandelt Funktionen als First Class Citizens

- „Basic“ Higher-Order Functions möglich
- Wenige Built-In Methoden
- Erweiterung durch Libraries

Python

- Behandelt Funktionen als First Class Objects
  - Python = alles ein Objekt

- „Basic“ Higher-Order Functions möglich
- Zusätzlich viele Built-in Methoden
- Function Decorator

Higher Order Functions in Python

- Function Decorator

```
def message_decorator(func):
    def wrapper():
        print(f"Before")
        func()
        print(f"After")
    return wrapper

@message_decorator
def function_decorator():
    print("Example function")

function_without_decorator =
    message_decorator(print("example function"))
```

Higher Order Functions in Python

- Map

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
```

- Filter

```
numbers = [1, 2, 3, 4]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
```

- Reduce

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
```

Higher Order Functions in Go

- Go Version <1.18:
  - eigene Higher-Order Funktionen für jede Datenstruktur

```
type Car struct {
    Name, Color string
}

//go:generate pie Cars.*
type Cars []Car
```

Library:
elliotchance/pie/v1

- Go Version >1.18 (2021):
  - Einführung Generics
  - Funktionale Programmierung lukrativer
  - Third-Party Library „lo“

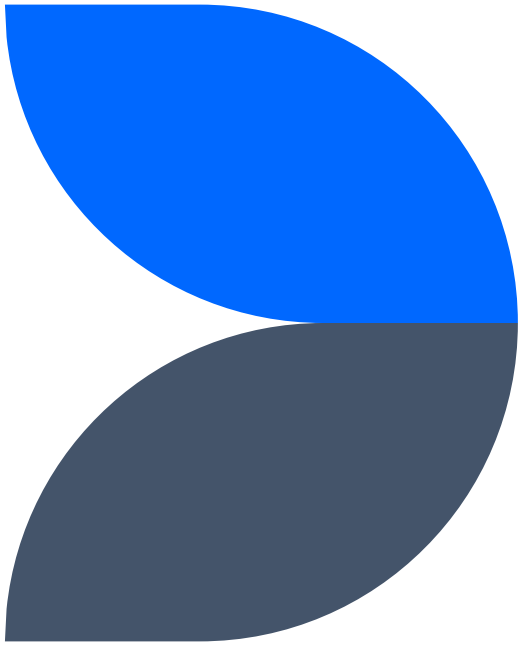
```
import "github.com/samber/lo"

lo.Map([]int64{1, 2, 3, 4}, func(x int64, index int) string {
    return strconv.FormatInt(x, 10)
})
// []string{"1", "2", "3", "4"}
```

  - Built-In Support für Higher Order Funktionen steigt: Slices + Maps

```
firstOldest := slices.MaxFunc(people, func(a, b Person) int {
    return cmp.Compare(a.Age, b.Age)
})
fmt.Println(firstOldest.Name)
```

Pure Functions



Pure Functions

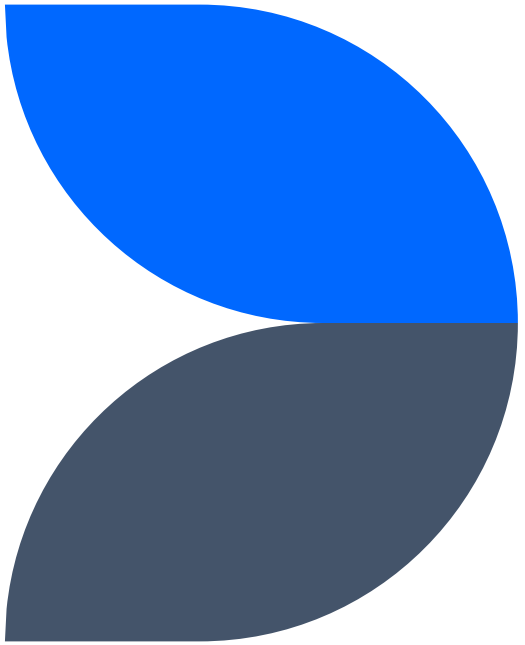
- Selber Rückgabewert für dieselben Inputwerte
- Keine Side effects

```
var globalVariable int
func rollDice() int {
    randomNumber := rand.Intn(6)
    globalVariable = randomNumber
    return randomNumber
}
```

Tipps für nicht rein funktionale Sprachen:

1. Vermeide globale Variablen/Zustände
2. Separiere pure und impure Funktionalität (API Calls, I/O Operations, ...)
3. ...

Immutability



Immutability

Go

- Mutable Datenstrukturen:
  - Arrays, Slices, Maps, Channels, eigene structs
- Immutable Datenstrukturen:
  - Strings, numeric values, interfaces ...
- Unterschied Pass By Reference und Pass By Value wichtig

- Map immer mutable!

Python

- Mutable Datenstrukturen:
  - List, Dictionary, Sets, eigene Klassen
- Immutable Datenstrukturen
  - Numbers, strings, tuples
- Definieren von immutable Klassen möglich (Pydantic, dataclasses, NamedTuple)
- Immutable Lists, Dictionaries etc.: pyrsistent

Immutability in Python

- Eigene Klassen

```
//example of Typings NamedTuple
class GameBoard(NamedTuple):
   game_board: Tuple[Tuple[Piece, ...], ...]
   currPlayer: str = "R"

//Example of Pydantic BaseModel
class GameBoard(BaseModel):
   game_board: Tuple[Tuple[Piece, ...], ...]
   currPlayer: str = "R"
   class Config:
      frozen = True
```

- Collections

```
>>> from pyrsistent import v, pvector

# No mutation of vectors once created, instead they
# are "evolved" leaving the original untouched
>>> v1 = v(1, 2, 3)
>>> v2 = v1.append(4)
>>> v3 = v2.set(1, 5)
>>> v1
pvector([1, 2, 3])
>>> v2
pvector([1, 2, 3, 4])
>>> v3
```

Immutability in Go

Pass By Reference/Value

```
func main() {
    names := []string{"Miranda"}
    addValue(names, "Yvonne")
    fmt.Printf("%v\n", names)
}
func addValue(s []string, name string) {
    s = append(s, name)
}
```

Output? → [Miranda]

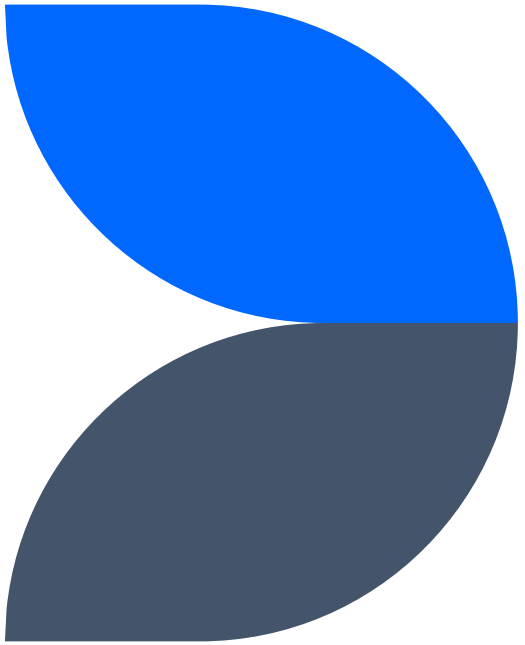
```
func main() {
    names := []string{"Miranda"}
    addValue(&names, "Yvonne")
    fmt.Printf("%v\n", names)
}
func addValue(s *[]string, name string) {
    *s = append(*s, name)
}
```

Output? → [Miranda Yvonne]

```
func main() {
    m := map[string]int{}
    addValue(m, "red", 10)
    fmt.Printf("%v\n", m)
}
func addValue(m map[string]int, colour string, value int) {
    m[colour] = value
}
```

Output? → [red 10]

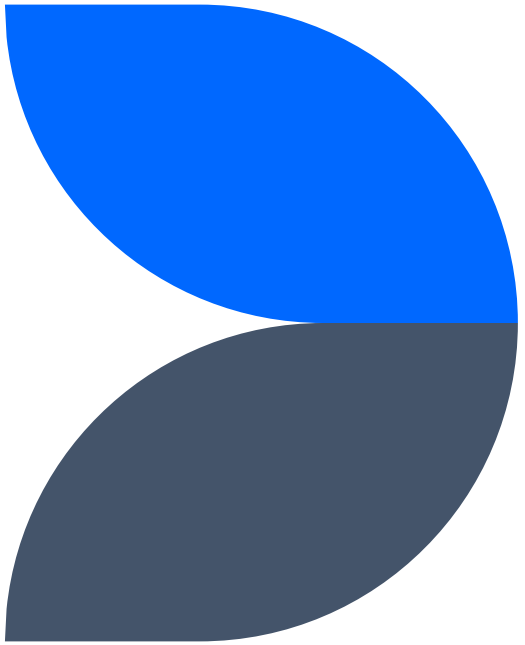
Recursion



Recursion

- Keine der beiden Sprachen ist optimiert für Rekursion
  - Langsamer als iterative Lösungen
  - Stackspeicher limitiert (Python: Stack Depth 1000 , Go: 1GB)
  - Keine Tail Call Optimization
- Bei rekursiven Algorithmen/Datenstrukturen das Mittel der Wahl

Lazy Evaluation



Lazy Evaluation

Go

- Go standardmäßig kein Lazy Evaluation
- Programmierer kann Funktionen lazy implementieren, wenn er darauf achtet

Python

- Support für Lazy Evaluation
  - Generator Konzept
- Viele Built-In Funktionen nutzen Lazy Evaluation out of the Box
  - Map, Filter, ...

Generator Functions Python

- Funktion gibt Generator (Spezialfall von Iterator) zurück
- Python Interpreter erkennt „yield“ und erstellt automatisch Iterator Objekt
- Berechnet werden die Werte erst, wenn sie benötigt werden.

```python
from typing import Generator

def generate_squares(limit: int) -> Generator[int, None, None]:
    for number in range(limit):
        yield number ** 2

squares_generator = generate_squares(5)

print("Generated squares:")
for square in squares_generator:
    print(square)
```

Comprehension Python

- Ohne Comprehension

- Mit List Comprehension [....]

- Mit Generator Comprehension (....)

```
//Solving without comprehension
collection = list()
for datum in data_set:
   if condition(datum):
      collection.append(datum)
   else:
      modify(datum)
```

```
//with comprehension
collection_list = [d if condition(d) else modify(d)
for d in data_set]
```

```
//with generator comprehension
collection_iter = (d if condition(d) else modify(d)
for d in data_set)
//consume iterator by client function
client = list(collection_iter)
client == collection_iter
// --> True
```

Figure 8: Convert imperative code to comprehension

Fazit

- Python mehr funktionale Konzepte als Go
- Beide Multiparadigma → also nicht rein funktional per Design
- Go wird / könnte mehr Unterstützung für Higher-Order Funktionen bekommen

Vielen Dank