FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

DIE HOCHSCHULE, DIE MEHR KANN.

ILV Einführung in das Programmieren

Funktionen

Mohamed Goha, BSc. | WS 2024/25

APPLIED LIFE SCIENCES | MASTERSTUDIENGANG BIOINFORMATIK

Motivation

> Oft kommen bestimmte Probleme/Aufgaben mehrmals vor

> Wir möchten die Folge an Operationen, die diese Aufgabe löst, mehrmals ausführen, manchmal mit verschiedenem Input

> Wir möchten Code duplication vermeiden

> Lösung: Funktionen

Beispiel: Berechnung des GC content

- Gleicher Code wird für jede Sequenz ausgeführt
- Nicht elegant und fehleranfällig

```
# For Sequence 1
sequence1 = "ATGCGCTA"
gc_count1 = sequence1.count("G") + sequence1.count("C")
gc_content1 = (gc_count1 / len(sequence1)) * 100
print(f"GC content of Sequence 1: {gc_content1}%")

# For Sequence 2
sequence2 = "ATTTAGCC"
gc_count2 = sequence2.count("G") + sequence2.count("C")
gc_content2 = (gc_count2 / len(sequence2)) * 100
print(f"GC content of Sequence 2: {gc_content2}%")
```

Funktionen

> Wir möchten den Code, der mehrmals vorkommt, extrahieren und **parametrisieren**

> Somit haben wir eine **Funktion**

```python
def calculate_gc_content(sequence):
    gc_count = sequence.count("G") + sequence.count("C")
    gc_content = (gc_count / len(sequence)) * 100
    return gc_content

# Now you can reuse the function with different sequences
print(f"GC content of Sequence 1: {calculate_gc_content('ATGCGCTA')}%")
print(f"GC content of Sequence 2: {calculate_gc_content('ATTTAGCC')}%")
print(f"GC content of Sequence 3: {calculate_gc_content('GGCCATTA')}%")
```

Vorteile

- Weniger code, gleiche Funktionalität
- Macht Code lesbarer
- Kopieren von Code-Abschnitten ist fehleranfällig
- **Bessere Wartbarkeit (code maintainability)**
  - Bei Fehlern und/oder Änderungen muss nur

    code in der Funktion einmal geändert werden!

Intuition Funktionen

> Vergleich mathematische Funktion

> Die Funktion **f** ist durch **x**

parametrisiert

```
# f(x) = kx + d
def f(x):
    k = 2
    d = 5
    return k * x + d
```

Funktionen in Python

> Funktionen werden durch das keyword **def** definiert und können **Parameter** und einen **Rückgabewert (output)** haben.

> Beide sind optional

> Namenskonvention ähnlich wie bei Variablen

Funktionen in Python: Beispiele

```python
def some_func():
    # do something
    # (no return, just side effects)

def some_func():
    # create a result and return it
    return result
def some_func(x):
    # do something with the argument
    # that is given for parameter x
    # (side effects)
def some_func(a, b, c):
    # do something with a, b, c
    # and return the result
    return a + b + c
```

Parameter und Argumente

> **Parameter** = Variablen, die in der Funktionsdefinition angeführt sind

> **Argument** = Werte, die man bei Funktionsaufruf den Parametern zuweist

```
def calculate_gc_content(sequence): # sequence is a parameter
    gc_count = sequence.count("G") + sequence.count("C")
    gc_content = (gc_count / len(sequence)) * 100
    return gc_content
gc_content = calculate_gc_content("ATGCGCTA") # ATGCGCTA  is the argument
```

Übergeben von Argumenten

> Ähnlich zu Variablenzuweisung

> Achtung bei mutable Objekten: wenn Objekt innerhalb Funktion geändert wird, dann gilt die Änderung auch außerhalb der Funktion!

```
def append_to_list(a_list, item):
    a_list.append(item)

some_list = [1, 2, 3]
append_to_list(some_list, 4)
print(some_list) # prints [1, 2, 3, 4]
```

Positional und Keyword Arguments

- Es gibt zwei Arten, Argumente zu übergeben:
  - Als **positional arguments**
  - Als **keyword arguments**
- Genaueres dazu folgt in der Übung

```
def append_to_list(a_list, item):
    a_list.append(item)

some_list = [1, 2, 3]

append_to_list(some_list, 4) # pos. args
append_to_list(a_list = some_list, item=4) # keyword args
print(some_list) # prints ... ?
```

Variable Argumente

- Es ist möglich, Funktionen so zu definieren, dass ihnen arbiträr viele argumente übergeben werden können.
- Für positional arguments können wir das tun, indem wir ***args** als parameter hinzufügen. (alle Argumente werden in einem tuple gesammelt)
- Für keyword arguments: ****kwargs** (alle Argumente werden in einem dictionary gesammelt)

Variable Argumente: Beispiel

```
def append_to_list(a_list, item, *args, **kwargs):
    a_list.append(item)
    print("args: ", args)
    print("kwargs:", kwargs)
some_list = [1, 2, 3]

append_to_list(some_list, 4, 5, 6, another_item = 7)
```

Parameter unpacking, Default Parameters, Type Hinting, Return Value

Siehe Input Übung 4!

Namespaces, Scopes

- Ein **Namespace** ist ein dictionary, das Variablen mit ihren Objekten assoziiert.
- Der **Scope** eines Namen (Einer Variable, etc) definiert seine look-up Reihenfolge (Wo dieser Name sichtbar ist)

Namespaces, Scopes

- **Namespaces**:
  - **built-in:** alle Python built-ins: len, dict, print, etc
  - **global**: namen, die im main script/module definiert sind (globale Variablen)
  - **local**: Namen, die im innersten Level definiert sind (z.B in einer Funktion)
  - **enclosing:** nested structures. z.B. die enclosing Funktion beinhaltet die Namen der inner function

Namespaces, Scopes: Beispiel

```
num = int("5")

def some_function(x):
    var = 70
    return x*var

# int(): part of built-in namespace
# num and some_function: part of global namespace
# x and var: part of local namespace
```