Programming Techniques I
Zusammenfassung

Philip Paul "Maul" Müller, Stefano Lorenz Weidmann, Ronan Jakob Lindörfer

June 8, 2017

Contents

1 git **3**
- 1.1 HEAD: . . . 3
- 1.2 Einrichten eines Resporitorys: . . . 3
- 1.3 Betrachten von Unterschieden: . . . 3
- 1.4 Branching: . . . 3
- 1.5 Reset: . . . 4
- 1.6 RM: . . . 5
- 1.7 Clone: . . . 5
- 1.8 Serververbindungen: . . . 5
- 1.9 Merge . . . 6

2 make **6**

3 cmake **9**

4 Python **11**
- 4.1 includieren . . . 11
- 4.2 Funktionen . . . 11
- 4.3 Copy . . . 14
- 4.4 Syntax . . . 14
  - 4.4.1 Vergleich Operatoren . . . 14
  - 4.4.2 if . . . 14
  - 4.4.3 while-Loop . . . 14
  - 4.4.4 For . . . 15
- 4.5 Datenstrukturen . . . 15
  - 4.5.1 List . . . 15
  - 4.5.2 Tuple . . . 16
  - 4.5.3 Dictionary . . . 16
- 4.6 Klassen . . . 16
  - 4.6.1 self . . . 18

5 Folien **19**

6 C++ **20**
6.1 Stochastische Methoden 20
6.1.1 Monte-Carlo-Integration 20
6.1.2 Importance sampling 21
6.1.3 Inversionsmethode 21
6.1.4 Verwerfungsmethode (Acceptance-Rejection-Methode) 21
6.2 Exeption 22
6.3 Funktions-Pointer 23
6.4 Simpson 24
6.5 BLAS 25
6.6 Vererbung (virtual) 26

1 git

http://swcarpentry.github.io/git-novice/

1.1 HEAD:

HEAD is eine Variable. Sie steht für den letzten Commit, der im aktuellen Branche gemacht worden ist. Dies bedeutet, dass head in jedem branch etwas anderes ist. Durch anhängen von n, (HEAD n) kann man n Commits zurückgehen.

1.2 Einrichten eines Resporitorys:

Bevor man git verwenden kann, muss man es einrichten. Zuerst setzt man seine Persönlichen Daten

//So wird der Name des Users gesetzt

```
git config --global user.name "Mein Name"
```

//So wird seine E-Mail-Adresse gesetzt

```
git config --global user.email meine.E@addresse
```

//So kann man den Editor, der git aufruft ändern

```
git config --global core.editor emacs
```

//Mit diesem Befehl wird die Ein lokales reporitory erstellt

```
git init
```

1.3 Betrachten von Unterschieden:

//Dies vergleicht das File mit dem "*Filename*" mit dem aktuellen Zustand des Files, durch angabe des *HACH* kann der Zeitpunkt bestimmt werden

```
git diff (HASH-Code) Filename
```

//Mit dieser Methode kann man beliebig im baum zurückgehen, n bezeichnet dabei die Anzahl Commits, wenn man nur HEAD schreibt, wird der letzte genommen

```
git diff HEADñ filename
```

//Mit diesem Befehl, kann man zwei beliebige Commits miteinander vergleichen

```
git diff <commit1> <commit2> [--] fileName
```

1.4 Branching:

Git kennt Branches, also Äste. Sie eigenen sich dazu, Dinge auszuprobieren. Das Tool, das git verwendet um brances zu managen, wird auch verwendet, um Filehistory zu verwalten. `checkout` wird verwendet, um auf den Branches zu navigieren. `branch` wird verwendet, um die Branches zu verwalten, zum Teil gibt es Überschneidungen.

//Wechseln vom aktuellen Branch zum Branch ziel, der existieren muss

```
git checkout ziel
```

//Erstellen des Branches ziel und anschliessendem wechseln zum Branch
```
git checkout -b ziel
```

//Erstellen eines Branches der von einem Remote abstammt
```
git checkout --track <remote/branch>
```

//Erstele den Branch ziel der direkt von der gegebenen CommitID abstammt.
```
git checkout -b ziel CommitID
```

Als Ziel kann auch ein Commit angegeben werden oder ein HEAD, der in der "Vergangenheit" liegt. Dies bewirkt, dass der Zustand, der durch diesem Commit repräsentiert wird, wiederhergestellt wird.

Mit Folgendem Befehl wird das File mit dem Namen *fname*, auf den Zustand, den es hatte als der Commit mit der CommitID erstellt wurde, es kann auch der Hash oder HEAD verwendet werden. Im Endeffekt wird das File wiederhergestellt. Die CommitID kann auch weggelassen werden, dann wird einfach der letzte Commit verwendet.

//Restore File von CommitID ("go back to")
```
git checkout CommitID fname
```

Git hat noch weitere Tools um mit Branches zu arbeiten. Branches werden mit dem Tool `branch` verwaltet.

//Löschen von einem Branch, wenn das d Gross is, ist es mit forse
```
git branch -d Name
```

//Erstellen eines Branches, der von HEAD erbt
```
git branch <Name>
```

//Auflisten aller aktuell existierender Branches
```
git branch -av
```

1.5 Reset:

Dies ist die Möglichkeit zu einem spezifischen Commit zurück zugehen, alles bis dahin wird gelöscht und ist unwiderruflich verloren. Es ist dadurch auch möglich Dateien aus der Staging-Area zu entfehrnen.

//Mit diesem Befehl kann ein File aus der Staging Area entfernt werden, die Änderungen bleiben aber erhalten
```
git reset -- Filename
```

//Entfehrnt das file aus der Staging Area und setzt es auf den letzten Commit zurück, Löscht Änderungen
```
git checkout HEAD file
```

//Mit diesem Befhel, geht man zurück zum Zustand mit der CommitID, Änderungen gehen verloren
```
git reset --hard CommitID
```

1.6 RM:

Dieser Befehl markiert die Datei mit dem Namen Filename für git als gelöscht. Dabei wird es nicht vom WD geslöscht, es wird in git einfach als gelöscht markiert.
`git rm Filename`

1.7 Clone:

Mit diesem Befehl kann man ein Remotresporitory klonen. Dabei wird automatisch (evtl.) ein Lokales Resporitory erstellt, auch wird automatisch das origin Objekt angelegt, das verwendet wird, um den Remote zu reprösentieren.
`git clone Addresse`

1.8 Serververbindungen:

Nun wird kurz erklährt, wie man eine lokales resporitory mit einem online resporitory verbindet. Der origin-Brance representiert, alles was auf dem Server ist. Er hat auch Brances, diesen können mit folgender Syntax angesprochen werden:
`origin/Brance_Name`

//Erzeugen eines "Branches", mit dem Namen origin der das entfernte Resporitory darstellt
`git remote add origin git@gitlab.phys.ethz.ch:.....`

//Mit diesem Befehl kann man, den *lokalen*-Branche ¡lbranche¿ auf den Remote ¡remote¿ schicken.
`git push <remote> <lbranche>`

Man kann aber auch Daten vom Remote hohlen. dabei gibt es zwei unterschiedliche Methoden, dies zu tun. `fetch` lädt die Dateien des Remotes, der git unter dem Name ¡remote¿ bekannt ist, nur herunter. ein Merge findet nicht statt. Die Branches usw. dieses Remotes sind dann via der Syntax: `<remote>/<Branche_Name>` zugänglich. Die Branches sind aber nicht direkt "da" sondern lediglich ein Verweis. Damit können sie in einen lokalen Branche gemerged werden.
`pull` macht eigentlich das gleiche, wie `fetch` es macht nur einen Merge. Die "Verlinkung" zwischen Branche, kann mit der u Option eingestellt werden.

//Mit diesem Befehl, lädt man den "Zustand" des Remotes ¡remote¿ herunter
`git fetch <remote>`

//Merge den Zustand des Remotebranch ¡rbranch¿ von Remote ¡remote¿ in den aktuellen Branch
`git merge <remote>/<rbranche>`

1.9 Merge

Das Arbeiten ist so. Man wechselt in einen Brance und macht dort was. Irgendwann ist man fertig und man will seine Arbeit, die man im Branche A gemacht hat, in den Branche B einpflegen. Hierzu wechselt man zuerst in den Branche B. Nun gibt man folgenden Code ein:
`git merge A`

Dieser Befehl fügt A in B ein. Wenn A & B einen gemeinsamen Vergänger haben, so kann git die vereinigung selber machen. Falls git Konflikte trifft, die es nicht auflössen kann, wird es beides einfügen und entsprechend kennzeichnen.

2 make

Anschliessen kommt hier ein make-File, das von Stefano Weidmann geschrieben wurde.

```
# use g++ if no compiler is defined
CXX ?= g++
# which standard to use in the default case
CXXSTD ?= -std=c++11

# and which flags
CXXFLAGS ?= -Wall -O2 -fPIC

# and which includes
CXXINCLUDES ?= -I. -I/usr/local/bin -I/opt/local/include

# and where to look for the libraries
CXXLIBS ?= -L/usr/local/lib -L/opt/local/lib

# headers which cause a rebuild
CXXHEADERS ?= /path/to/frail/header/likely/to/change.hpp

# install prefix
INSTALLPREFIX ?= /usr/local

compile := $(CXX) $(CXXSTD) $(CXXFLAGS) $(CXXINCLUDES)

# output all variable values, only useful for debugging
varsToPrint := CXX CXXSTD CXXFLAGS CXXINCLUDES CXXLIBS CXXHEADERS
$(foreach var,$(varsToPrint),$(info $(var) = $($(var))))

# the first target make encounters is its default target
# --> here: data
# By default, generate some data
data: simulation
./simulation > data

# link the simulation
```

```
simulation: libintegrate.a libdynamic.so simulation.o
$(compile) simulation.o $(CXXLIBS) -ldynamic -lintegrate

# Build required libraries:
# one static and one dynamic
libintegrate.a: integrate.o
ar ruc $@ $<
ranlib $@

libdynamic.so: dynamic.o
$(compile) -shared -o $@ $<

# rule how to make any object file blabla.o from its corresponding cpp file blabla.cpp
# recompiles if any headers changes
%.o: %.cpp $(CXXHEADERS)
$(compile) -c $< -o $@

# The ".PHONY" line makes the clean rule apply even if by chance there is a file
# named 'clean'
.PHONY: clean install
clean:
rm *.o

bindir := $(INSTALLPREFIX)/bin
libdir := $(INSTALLPREFIX)/lib
incldir := $(INSTALLPREFIX)/include
installflags := --compare --backup=existing --suffix=.bak
install: simulation
install --directory $(bindir) $(incldir) $(libdir)
install $(installflags) --strip simulation $(bindir)
install $(installflags) libintegrate.a libdynamic.so $(libdir)
install $(installflags) $(CXXHEADERS) $(incldir)

# if you have any problems with your rules not being applied, they could by
# shadowed by a
# builtin rule, so try adding this:
.SUFFIXES:

################################################################################
#                                THEORETICAL STUFF                             #
################################################################################

# common predefined (called "automatic") variables
# ================================================
# $@ The target of a rule
# $< The first dependency of a rule
# $^ All dependencies of a rule, separated by spaces
# $? All dependencies of a rule that are newer than the target, separated by spaces
```

```
# terminology
# ===========
# a rule looks like this
# target: dependency1 dependency2 ...
# \tshellcommand1
# \tshellcommand2
#...
# where \t is the tab character
# The shell commands have to be indented and the indent has to be a tab
# character, not spaces!

# example
# =========
# in the rule
# simulation: libintegrate.a libdynamic.so simulation.o
# $(compile) simulation.o $(CXXLIBS) -ldynamic -lintegrate
#
# simulation is the target and libintegrate.a, libdynamic.so, simulation.o
# are the dependencies, the automatic variables have the values:
# $@ = simulation
# $< = libintegrate.a
# $^ = libintegrate.a libdynamic.so simulation.o
#
#
# calling make from the shell
# ===========================
# Variant 1: make
# builds the first target in encounters in a makefile called 'Makefile' or similar
# in the current directory
#
# Variant 2: make simulation
# builts the target called 'simulation' in a makefile called 'Makefile' or similar
# in the current directory
#
# Variant 3: make -f /path/to/other/makefile.make
```

3 cmake

Hier ist ein Beispiel cmake-File abgedruckt, das so ziehmlich alles kann, was wir können müssem

```
cmake_minimum_required(VERSION 2.8)

project("My little project")

# Call as cmake -G 'Unix Makefiles' -B"${build directory}" -H"${source directory}"
# or with build directory as the working directory
# cmake "${source directory}"
# then a Makefile will be generated.
# Proceed with make && make install

# add include directories
include_directories("/opt/local/include" /usr/local/include
"${CMAKE_SOURCE_DIR}" /here/are/my/headers)

# ${CMAKE_SOURCE_DIR} points always to the directory in which the first CMakeLists.txt-File
# in subdirectories it still points to the folder above.


# Set the type of the build of the program
set(${CMAKE_BUILD_TYPE} type)
# types are:  None Debug Release RelWithDebInfo MinSizeRel


# Require C++ 11
# DO THIS BEFORE ANY add_executable OR add_library
set(CXX_STANDARD_REQUIRED on)
set(CXX_STANDARD 11)

# Seting the compiler flags
# Used in this way, it gets expanded
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")

# Set the default instalations dir
set(CMAKE_INSTALL_PREFIX /path/to/default/install/dir )

# Build foo from foo.cpp and bar from bar.cpp
set(programs foo bar)
foreach(prog ${programs})
   add_executable(${prog} ${prog}.cpp)
endforeach()

# build qux using more than one dependency
add_executable(qux qux.cpp otherDependency.cpp)

# build a static library
```

```
add_library(myIntegrationLibrary STATIC myIntegrationLibrary.cpp)

# build a dynamic library
add_library(myDynamicLib SHARED myDynamicLib.cpp)

# link executable
target_link_libraries(qux MyIntegrationLibrary myDynamicLib)

# install targets to somewhere
install(TARGETS foo bar qux
        ARCHIVE DESTINATION /path/for/static_libs   # Absolute path
        LIBRARY DESTINATION .                       # Directly in the install Root
        RUNTIME DESTINATION path/for/binaries)      # without a leading slash, cmake
                                                # interprets the path as a relative one,
                                                # wrt the CMAKE:INSTALL_PREFIX

# install other files
install(FILES myPrettyHeader.hpp DESTINATION /path/for/includes)

# build also stuff in subdirectories
set directoriesToProcess(this and that)
foreach(dir ${directoriesToProcess})
   add_subdirectory(${dir})
endforeach()
```

4 Python

In dieser Sektion gibt es eine Kurze Abhandlung über Python. Python hat keine Typen, obwohl es eigentlich schon welche hat, sie sind einfach vom Benutzer "verborgen".

4.1 includieren

Python bietet die Möglichkeit beim inkludieren einen Namensraum zu erzeugen. Es gibt verschiedene Möglichkeiten ein Importstatement hinzuschreiben und alle führen zu etwas anderen Ergebnissen. Generell besteht das Importstatement aus folgenden Keywords `import, from, as`, wobei einige optional sind.

from

Mit dieser Syntax könen nur gewisse Funktionen und oder Klassen in das aktuelle File importiert werden

```
from filename import f as r
```

Diese Zeile bewirkt, dass die Funktion/Klasse f vom file `filename.py` importiert wird. Man kann auch mehrere Dinge importieren, indem man eine Komma getrennte Liste mit den zu importierenden Funktionen angibt.

Ist das File `filename.py` nicht im gleichen Ordner, sondern im unterordner `include`, so kann man die Hierarchie durch Punkte angeben. Dies sieht wie folgt aus.

```
from include.filename import f, g
```

Diese Linie importiert die "Objekte" f und g aus dem File `./include/filename.py`. Die Funktionen werden in den Globalen Namensraum importiert.

Wenn man alles aus dem File importieren möchte so schreibt man einen $*$ anstatt alle aufzulisten.

import

Will man hingegen ein ganzes "Packet" importieren, so muss man folgende Syntax verwenden

```
import filename as alias
```

Diese Zeile importiert alles was, im File mit dem Namen `finename.py` und macht es im "Namensraum" Alias zugänglich. Auch hier gelten die Regeln mit dem Ordnern.
Angenommen in besagtem File hat es eine Funktion f(x,y), dann ist diese Funktion mit folgender Syntax zugänglich.

```
alias.f(x,y)
```

4.2 Funktionen

Wie andere Sprachen auch, kennt Python Funktionen. Sie haben aber einige Besonderheiten.

- Es wird zwischen *positional* arguments und *keyword-only* arguments unterschieden
- Argumente können gepackt und ungepackt werden
- Die Reihenfolge, im Aufruf, kann *beliebig* geändert werden.

Die allgemeine Definition, einer Funktion sieht zu aus:

```
def func(x, y=1, *args, z, a= 1, b, **kwargs)
```

`x`	**positional** arguments
`y`	positional arguments with defaults
`args`	**variadic positional arguments**
`z, b`	**keyword-only** arguments
`a`	**keyword-only** arguments, with **defaults**
`kwargs`	**variadic** keyword-only arguments

Positional Arguments

Diese Argumente sind wie die normalen C++-Argumente. Eine Funktion kann aufgerufen werden, ohne das die "Interna" der Funktion, sprich die Bezeichnung der Argumente in der Funktionsdefinition bekannt sein müssen. Bei der Dekleration der Funktion können mit dem =-Zeichen, den Argumenten Default-Werte gegeben werden, sie müssen nach den Argumenten kommen, die keinen Default-Werte haben. Die Zuordnung erfolgt aufgrund der *Reihenfolge*. Jedoch ist es *optional* möglich, dass man die "Argumente "benannt" aufruft. Dies ermöglicht, dass die Reihenfolge, beliebig gewählt sein kann.

```
def f(x, y = 1, z = 2):
        # Definition of f

p = Person()

# Call with default
f(p)  # x = p; y = 1; z = 2

# Still same order
f(p, 2)  # x = p; y = 2; z = 2

# Call full
f(p, -3, -9)  # x = p; y = -3; z = -9

# interchange order
f(y = 3, x = p, z = 4)  # x = p; y = 3; z = 4
```

*args - variadic positional arguments

"variadic positional arguments" bezeichnet Argumente, deren *Anzahl* zum Zeitpunkt der Definition der Funktion unbekannt ist. Ein Beispiel für eine solche Funktion ist die `printf`-Funktion von C. Sie kommen in der Definition, nach den positional arguments und wird durch eine Variable, die mit einem * beginnt, gekennzeichnet. Aus Konvention, nennt man diese Variable `args`.
Das Konstrukt `*args` hat je nach Ort, wo es steht eine unterschiedliche Bedeutung.

In der **Definition** bedeutet `*args` das ein ein *Tupel* gibt das `args` heisst. Im Aufruf bedeutet es, dass man ein Tupel machen will. Ein Beispiel.

```
def f(*args):
        # Definition of f

# Definition of some variables
```

```
x = 1
y = 2
z = "Hans"

f( *(x,y,z))
# Now args is a tuple that looks like args = (1, 2, "Hans")

f( *(2, "Berta", 4) )
# Now args is a tuple that looks like args = (2, "Berta", 4)
```

Keyword-Only Parameter

Diese Parameter können *nur* mit ihrem Namen aufgerufen werden. Man muss also wissen, wie die Definition der Funktion genau aussieht. Auch können sie optional Default-Werte haben. Diejenigen Parameter die Default-Werte haben, sollten nach denjenigen ohne kommen. Da man sie nur mit ihrem Namen aufrufen kann, ist ihre Reihenfolge beim Aufrufen der Funktion egal. Der Interpreter unterscheidet sie von den Positional Arguments dadurch, dass sie *nach* dem `*args` kommen.

```
# This function has only keyword-only parameters
def f(*, a, b = 3):
        # Definition of f

f(a = 2)  # a = 2; b = 3

f(2)  # ERROR: because keyword-only

f(b = 1, a = 5)  # a = 5; b = 1
```

Vardiac Keyword-Only Parameter

Sie funktionieren ähnlich wie die Vardiac's bei den Positional Arguments. Sie werden durch zwei $*$ gekennzeichnet und aus Konvention nennt man die Variable `kwargs`. Der unterschied besteht darin, dass sie nicht als Tupel gepackt werden, sondern als "Dictionary", was im wessentlichen einer `std::map` aus C++ entspricht. Die Schlüssel sind dabei die Namen die gewählt worden sind, um sie zu benennen. Dabei können Phantasie Namen gebraucht werden. Die Regel scheint so zu sein: *Alles was man am Ende des Aufrufs der Funktion, nicht zuordnen kann ist Teil des Vardiac KW-O Argumentes.*
Mischen ist Gefährlich.

```
def f(x, y = 1, **kwargs):
        print(kwargs, type(kwargs))

# Call the funtion in a normal fashion
f(1, y=2, a = 3, z = 4)
# {'a' : 3, 'z' : 4} <class 'dict'>

# Equivalent methods, that do the same as above
f(**{'x' : 1, 'y' : 2, 'a' : 3, 'z' : 4})
f(1, 2, a = 3, **{'z' : 4})
```

4.3 Copy

Aufgrund der komischen Speicherverwaltung von Python, die vermutlich hin und wieder sehr effizient sein kann, ist Kopieren kritisch. Man sollte demnach immer eine *Deepcopy* machen.

```
import copy

x = [0,1,2]

# Perform a DEEPCOPY, this should be the DEFAULT
y = copy.deepcopy(x)  # Real copy of x


# Perform a shallow copy
b = copy.copy(x)
```

4.4 Syntax

Nun folgt ein kleiner Überblick über die Syntax von Python. Generell ist sie ähnlich mit der von C++, weicht jedoch manchmal stark von ihr ab.

4.4.1 Vergleich Operatoren

Logische Operatoren in Python werden nicht durch die Symbole && und || dargestellt, sondern durch die Wörter `and, or, not` dargestellt.
Der logische Wert `Wahr` wird durch `True` und `Falsch` entsprechend als `False` dargestellt.
Die Vergleichs Operatoren sind die gleichen wie in C++.

4.4.2 if

Das if Statement ist auch ähnlich wie in C++. Es gibt aber noch zusätzlich ein `elif`-Statement. Die Syntax ist die folgende:

```
if PREDICATE:
        # If PREDICATE evaluates to True,
        # do this
elif PREDICATE2:
        # if PREDICATE2 evaluates to True,
        # do this instead
else:
        # if none of the above is True,
        # do this
```

4.4.3 while-Loop

Der while-Loop funktioniert gleich wie in C++, auch kennt Python die Befehle `break` und `continue`.

```
while PREDICATE:
        # Do something
        if PREDICATE2:
                # If PREDICATE2 is True, stop the loop
                break
```

```
        elif PREDICATE3
                # If PREDICATE3 is True, continue the loop
                continue
        # Do more stuff
```

4.4.4 For

Der for-Loop ist nicht direkt vergleichbar mit dem im C++, er funktioniert anders. Generell hat er zwei Modies. Im einten imitiert er den C++ loop, im anderen kann man durch eine Liste iterieren.

Dies ist der C++-Modus

```
for it in range(a, b):
        # Do something with it
        # it goes from a up to b-1
```

Nun kommt der "Iterator" - Modus

```
# let a be a list
for member in a:
        # Do something with member
        # member will go throught all members of a
```

4.5 Datenstrukturen

Python kennt verschiedene Datentypen und Strukturen. Allerdings ist die "Wahl" nicht fest. Einer Variable der ein int zugewissen worden ist, kann später ein String oder was anderes zugewiesen werden.

Daneben kennt Python noch einige eingebaute komplexere Datenstrukturen.

ACHTUNG:
Python hat eine sehr komische Speicherverwaltung. Wenn man etwas kopieren muss, so muss man das Copy-Packet nehmen. Siehe hierzu 4.3.

4.5.1 List

Die `list` ist mit dem `std::vector` vergleichbar. Man kann über sie iterieren und auf Elemente zugreifen. Wie man über sie iteriert ist im Abschnitt 4.4.4 erläutert. `list` können mit dem `range(a, b)` erstellt werden, der eine Liste mit den $\{a, a+1, \ldots, b-1\}$ erzeugt. Sie kann aber auch durch Angabe, der Elemente definiert werden. **Es ist wichtig zu sagen, dass die Elemente in der Liste nicht den gleichen Type haben müssen.**

```
# Create a list with name a
a = [1,2,3,4,5]

# Accass Element
a[2] #  yiels 3

# remove the element at position 3and return it
a.pop(3) #  yiels 4; a = [1,2,3,5]

# insert a element 7 before the element at position 1
a.insert(1, 7) #  a = [1,7,2,5]  'index, value'
```

```
# append the element 8 at the end of the list
a.append(8) #  a = [1,7,2,5,8]
# Equivalent to a.index(len(a), 8)


# Remove the first element in the list, that is equal to 5
a.remove(5) #  a = [1,7,2,8]

# Append the list ['tree', 4] to a
a += ['tree', 4] #  a = [1,7,2,8,'tree', 4]
```

4.5.2 Tuple

Ein Tupel ist eine Liste mit *unänderbaren* Objekten. Sie kann im wesentlichen alles was eine Liste kann, solange dies das Objekt nicht verändert. Jedoch können sie änderbare Objekte beinhalten, wie Listen. Sie wird auch anders Definiert.

```
# Create a Tuple
a = (1, 'a', 5)

# Access a element
a[0]  # yiels 1
a[1]  # yiels 'a'

# No modifing is allowed
a[2] = 8  # Raise a Error
```

4.5.3 Dictionary

Ein `Dictionary` ist mit einer `std::map` vergleichbar. Sie ist ein assoziatives Array. Sie werden durch durch die Syntax { key : Value } erzeugt.

4.6 Klassen

Klassen verwenden das sogenannte "Ducktyping", dies bedeutet,

"When I see a bird that walks like a duck and swim like a duck and quacks like a duck, I call that bird a duck."

Die Members der Klassen, müssen *nicht* explizit angegeben werden, sondern können *dynamisch* hinzugefügt werden.

Die Operatorüberladung wird durch magic-words erreicht. Das wichtigste davon ist `__init__`, dass in etwa mit dem Konstruktor zu vergleichen. Die "Standard" Definition einer Klasse ist.

```
class Person:
        # "Default" constructor
        def __init__(self, x = 5.0, name = "Hans-Jakobli", AHV_Num = 485161):
                self.m_x = x
                self.m_name = name
                self.m_AHV_Num = AHV_Num
                # Do some more useful stuff
```

```
        def __str__(self):
                s = "N: {0}\nAHV: {1}".format(self.m_name, self.m_AHV_Num)
                return s

        def name(self):
                return self.m_name

        @property
        def name2(self):
                return self.m_name

        def ReName(self, value):
                # Do something
                self.m_name = value

        @reName.setter
        def reName(self, value):
                # Do something
                self.m_name = value
```

Die Commands, die mit einem @-Zeichen beginnen, sind sogenannte `Decoraters`, sie bieten "syntactic sugar".

`@property` und `@{FunktionsName}.setter` erlauben den Elementzugriff zu abstrahieren. Von Aussen sieht es so aus, als ob man auf einen Member zugreift, jedoch wird eine Funktion ausgeführt. Der folgende Code erklährt es.

```
import Person as Pers
Hans = Pers() # Hans: m_x = 5.0; m_name = "Hans-Jakobli"; m_AHV_Num = 48161
Rossi = Pers(name = "Rossi") # Hans: m_x = 5.0; m_name = "Rossi";
                             # m_AHV_Num = 48161

newName = "Frieda"

Hans.name()
# >>> "Hans-Jakobli"

# No parantese needed due the property
Hans.name2
# >>> "Hans-Jakobli"

# Change the value of the m_name member, by using a setter
Rossi.reName = newName
str(Rossi)
# >>> Name: Frieda
#     AHV-Nummer: 48161

# Change the value of the m_name member, by a function
# (classical setter function)
Rossi.ReName("Prometheus (Teusi)")
Rossi.name2
# >>> "Prometheus (Teusi)"
```

4.6.1 self

Das `self`-Keyword hat die selbe Bedeutung wie `this` in `C++`. Es muss aber in Python, *explizit* angegeben werden. Der "Nebeneffekt" ist, dass man *jedes* Wort nehmen kann. Python interpretiert, die erste Variable, in den Argumentklammern der Memberfunktion, einfach als die "`self`"-Variable.

Beim Aufruf kann sie dann weggelassen werden. Die Zählung beginnt also bei der zweiten Variable.

5 Folien

In diesem Abschnitt, wird zusammengefasst, was in den Vorlesungsfolien wo ist. Die Namen beziehen sich auf die Namen der Files, die sie uns gegeben haben. Die Reihenfolge, in der die Themen aufgezählt werden, korrespondiert in etwa mit der Reihenfolge ihres Auftretens in der Folie.

week1a Einfaches C++ Zeug, wie Schlaufen, Datentypen usw.
Aber auch Dinge wie Arrays und Pointers.
Sowie "Pass by Value" und "Pass by Reference".
Sowie etwas über Namensräume.

week1b Im wesentlichen "Version Controll".

week2a C-Preprocessor.

week2b cmake

week3a Ist in zwei Teile aufgeteilt.
"Wie funktioniert die CPU".
"Wie funktioniert das RAM".
Sowie etwas über den Cache.

week3b Eine kleine Einführung in die Template Programmierung.
Warum `C-makros` Gefährlich sein können.
Theorie, warum Templates gut sind.

week4 Eine kleine Einführung in Klassen.
Eine kleine Erläuterung des Penna-Modells.
Wie designt man eine Klasse.
Ein kleines Beispiel anhand einer Verkehrsampel.
Zugriff auf die Mitglieder der Klasse, das Zugriffsmodell.
Spezielle Mitglieder, wie der Konstruktor.
Ein kleines Beispiel anhand eines Punktes in 2D.
Der Unterschied von Dekleration, Definition und Implementation (Seite 12).
`Const`, `volatiole` und `mutable`.
Befreundete Funktionen.
Inlining und statische Funktionen.
Class-Templates.

week5 Weiter mit Klassen nun geht es um Operatorüberladung.
Auch um Konversion von Objekten.
Auf Seite 5 ist die Liste mit den automatischen Konversionen.
Template-Specialization und Type-Traits und natürlich die Verwendung von typename.
Erläuterungen von Konzepten.
`Koenig Lookup`, auf Seite 13.
Generische Programmierung anhand der Power-Methode.
Funktionsobjekte.

week67 Algoritmen und Datenstruckturen in C++.
Laufzeitanalyse.
Die STL.

Iterators, mit ganzem generischen Konzept.

week8 Monte Carlo Integration and Random Numbers.
Linear Congruential Generators $x_{n+1} = (ax_n + c) \mod m$.
Lagged Fibonacci Generators $x_n = x_{n-p} \otimes x_{n-q} \mod m$.
Inverse transform method.
Acceptance-rejection method.
Zufallszahlen in C++, Seite 10.

week9 Vererbung und Ausnahmen in C++.
Vererbung am Beispiel von Penna.
Abstrakte Basis-Klassen.
Virtuelle Funktions Tabelle.
Vergleich von Virtuellenfunktionen mit Templates.
Ausnahmebehandlung, Seite 8.
Programmierparadigmen in C++.
Objektorientiertes Programmieren.

week10 | **Week 11** To code or not to code (Optimierng Teil 1).
Profiling.
Die Wahl der Datenstruktur.
Strassen-Algorithmus.
Wie Optimiert man?.
Compiler intrinsic.
Was kann der Compiler selber machen.
Loop unrolling.
Aufrufen von FORTRAN Code aus C++.
BLAS.

week11 | **Week 12** To code or not to code (Optimierung Teil 2)
Metaprogramming, aka. auswertung in der Übersetzungszeit. Am Beispiel des Skalarproduktes.
Expression Templates.
Super duper abstrakte Version von Expression Templates von Vektoren.
TynyVector.

week11b Wie benutze ich make.

week12 Python.

6 C++

6.1 Stochastische Methoden

6.1.1 Monte-Carlo-Integration

Wir wollen ein Volumenintegral $I := \int_\Omega g(y)\, d\Omega$ mit $\Omega \subseteq \mathbb{R}^d, g : \Omega \to \mathbb{R}$ berechnen.
Idee: Wir fassen Y als über Ω gleichverteilte Zufallsvariable auf. Dann ist die Dichte von Y gegeben als $f_Y(y) = \frac{1}{\int_\Omega d\Omega} =: \frac{1}{V}$.
Nun betrachten wir den Erwartungswert $E(g(Y)) = \int_\Omega f_Y(y) \cdot g(y) d\Omega = \frac{\int_\Omega g(y)\, d\Omega}{V} = \frac{I}{V}$. Diesen

nähern wir mit dem Gesetz der grossen Zahlen an und erhalten $I = E(g(Y)) \cdot V \approx \frac{V}{n} \sum_{i=1}^{n} g(y_i)$ mit $Y_i \sim \text{Uni}(\Omega)$, i.i.d., n sehr gross. Dieses Verfahren heisst *Monte-Carlo-Integration (mit gleichverteiltem Sampling)* und hat einen stochastischen Fehler von $\Delta = \sqrt{\frac{\text{Var}(g(Y_i))}{n-1}} \propto \frac{1}{\sqrt{n}}$.
Für $d > 8$ Dimensionen ist es besser als die Simpsonregel.

6.1.2 Importance sampling

Wenn $\text{Var}(g(Y_i))$ gross ist, kann der Fehler mit gleichverteiltem Sampling sehr gross werden. Dann lohnt es sich, nicht gleichverteilte Y_i zu wählen, sondern Zufallsvariablen Z_i mit einer anderen Verteilung $Z_i \sim D(\ldots)$ und Dichte $f_Z(z)$. Das geht mit dem Trick $h(z) := \frac{g(z)}{f_Z(z)}$. Denn so ist $I = \int_\Omega g(y)\, d\Omega = \int_\Omega h(z) \cdot f_Z(z)\, d\Omega = E(h(Z))$.
Wie vorhin nun $E(h(Z)) \approx \frac{1}{n} \sum_{i=1}^{n} h(z_i)$ mit $Z_i \sim \text{D}(\ldots)$, i.i.d., n sehr gross.
Nun hat man aber den Vorteil, dass der Fehler von $\text{Var}(h(Z))$ abhängt. Wenn man also $f_Z \approx g$ wählt, ist $h \approx$ const. und deshalb $\text{Var}(h(Z)) \approx 0$ und der Fehler wird sehr klein.
Anmerkung: Wählt man wieder $f_Z(z) := \frac{1}{V}$, landet man wieder bei gleichverteiltem Sampling.

6.1.3 Inversionsmethode

Man hat eine Verteilungsfunktion F zu einer Verteiling $V(\ldots)$ gegeben und will nun Zufallsvariablen Y_i realisieren, die dieser Verteilung gehorchen.
Dazu realisiert man Zufallsvariablen $X_i \sim \text{Uni}([0,1])$. Dann sind $F^{-1}(X_i) \sim V(\ldots)$

Beispiel:

```
#include <random>
// C++ 11 benötigt

double F_inverse(double x)
{
   // ...
}

using RNG = std::mt19937_64;
const RNG::result_type seed = 42;
RNG generator(seed);

std::uniform_real_distribution<double> uni(0.0, 1.0);
double F_distributed = F_inverse(uni(generator));
```

6.1.4 Verwerfungsmethode (Acceptance-Rejection-Methode)

Ist es impraktikabel F^{-1} zu berechnen, verwendet man statt der Inversionsmethode die *Verwerfungsmethode*.
Zu einer gegebenen Verteilung $V(\ldots)$ mit Dichte $f(x)$ nimmt man eine einfach zu berechnende Hilfsverteilung $W(\ldots)$ mit Dichte $g(x)$. Diese Dichte muss ausserdem $f(x) \leq k \cdot g(x)$ für ein $k \in \mathbb{R}$ und alle $x \in \mathbb{R}$ erfüllen. Dann realisiert man $Z_i \sim W(\ldots)$ und $X_i \sim \text{Uni}([0,1])$. Wenn gilt $x_i < \frac{f(z_i)}{k \cdot g(z_i)}$, so ist z_i auch eine $V(\ldots)$-Zufallszahl (accept). Ansonsten generiert man zwei neue x_{i+1}, z_{i+1} (reject).

Beispiel:

```
#include <random>
// C++ 11 benötigt

// Dichte der zu realisierenden Verteilung
double f(double x)
{
   // ...
}

// Dichte der Hilfsverteilung
double g(double x)
{
   // ...
}

// Hilfsverteilung
using GDistribution = // ...;
GDistribution gDistrib(...);

// Hilfskonstante
const double k = // ...;

using RNG = std::mt19937_64;
const RNG::result_type seed = 42;
RNG generator(seed);

using UniDistribution = std::uniform_real_distribution<double>;
UniDistribution uni(0.0, 1.0);

double x; // Soll V-verteilt sein
double u; // Wird Uni([0, 1])-verteilt sein

do
{
   x = gDistrib(generator);
   u = uni(generator);
} while (u * k * g(x) >= f(x));

// x ist jetzt V-verteilt
```

6.2 Exeption

In C++ gibt es mehrere Header die eine Exepion zur Verfügung stellen. Generell gilt, dass alles gewofen werden können. Im Header `<exception>` gibt es eine Basis-Klasse von der die anderen Exeption erben. Im Header `<stdexcept>` gibt es verschieden Klassen. Allen diesen Dingern ist gemein, dass

der Konstruktor ein String frisst und eine Funktion Namens `what()`, die den String ausgibt. Mit der Referenz kann die Vererbung ausgenutzt werden. Es können auch mehrere `catch`-Blöcke nacheinander kommen. Wenn dies der Fall ist, geht der Compiler, alle von *oben nach unten* durch und stoppt beim *erst besten*, auch Vererbung.

```
#include <iostream>
#include <exception>
#include <stdexcept>

void f()
{
//... Do something
throw std::logic_error("Something goes Wrong");
}

int main()
{
try
{
f();
} //Catch an exepteion of type std::logic_error
 catch(std::logic_error& e)
 {
    std::cout << e.what(); // prints "Something goes Wrong"
 }
 //Catch all other exeption that inherented from std::exeption
 catch(std::exception& e)
 {
    std::cout << e.what();
 }
 //Catch the rest
 catch(...)
 {
    //Do something
 }

 return 0;
};
```

6.3 Funktions-Pointer

Funktionspointer sind Pointer auf Funktionen. Im Endeffekt können sie gleich Aufgerufen werden wie eine Funktion, haben aber eine komische Signatur. Die Signatur hat die folgende Gestallt `RetType(*Name)(Arg`
Ein Beispiel

```
//Function Prototypes
int foo();
double bar(int);
```

```
//FunctionPointers
int (*Pt1)() = foo;  //Pt1 zeigt nun auf foo
double (*Pt2)(int) = bar;  //Pt2 zeigt nun auf bar

//Aufrufen
int i = (*Pt1)();  //Aufruf via explizieter derefernzierung
double h = Pt2(i); //Aufruf via implizieter dereferenzieung

//In Funktionssignatur
int fancyFunc(double* Array, bool (*compFkt)(double, double))
{
//Do something
}

//Aufruf
double B[10];
fancyFunc(B, std::less<double>);
fancyFunc(B, &std::less<double>); //So kann man es auch noch machen
    //Solange die funktion eindeutig bestimmt werden kann

//Template
Die Spezialisierung eines Templates sieht so aus
template<typename R, typename D>
int fancyFunc(D*, R(*)(D));
```

6.4 Simpson

Das Simpson-Methode dient zur numerischen Integration. Hier ist ein kleines Beispiel.

```
#include <iostream>
#include <cmath>
#include <cassert>

double integrator(const double a, const double b, const unsigned int bins, double(*func)(do
{
    assert(bins > 0);

    typedef double domain_t;
    typedef double result_type;
    const domain_t dr = (b - a) / (2.*bins);
    result_type I2(0), I4(0);

    for(unsigned i = 0; i < bins; ++i)
    {
        domain_t pos = a + (2*i+1.)*dr;
        I4 += func(pos);
        I2 += func(pos+dr);
    }
```

```
    return (func(a) + 2.*I2 + 4.*I4 - func(b)) * (dr/3.);
}; //End of function integrator


int main()
{
    //...
    double Result  = integrator(a, b, n, &myF);
    double Result2 = integrator(a, b, n,  myF);

    //...

    return 0;
};
```

6.5 BLAS

```
// Kompilieren mit g++ -std=c++11 blasTest.cpp -lgfortran -lblas

#include <iostream>
#include <algorithm>

void output(const double* M, const int m, const int n) {
   for (int i(0); i < m; ++i) {
      for (int j(0); j < n; ++j) {
         std::cout << M[i*m+j] << " ";
      }
      std::cout << "\n";
   }

   return;
}
extern "C" void dgemm_(char* transA, char* transB, int* m, int* n, int* k,
              double* alpha, double* A, int* lda,
              double* B, int* ldb, double* beta,
              double* C, int* ldc);
int main() {
   double A[4];
   A[0] = 1;
   A[1] = 0;
   A[2] = A[1];
   A[3] = A[0];
```

```
   double B[4];
   B[0] = 1;
   B[1] = 2;
   B[2] = 3;
   B[3] = 4;

   double C[4];

   // BLAS OPTIONS
   char transA('T');
   char transB('T');
   int m(2);
   int n(2);
   int k(2);
   double alpha(1);
   double beta(1);
   int lda(k);
   int ldb(n);
   int ldc(m);

   // ATTENTION: BLAS IS COLUMN MAJOR
   dgemm_(&transA, &transB, &m, &n, &k, &alpha, A, &lda, B, &ldb, &beta, C, &ldc);
   // transpose C
   std::swap(C[1], C[2]);
   output(A, 2, 2);
   std::cout << "\n*\n";
   output(B, 2, 2);
   std::cout << "\n=\n";
   output(C, 2, 2);

   return 0;
}
```

6.6 Vererbung (virtual)

```
#include <iostream>

class objekt{
   public:
   objekt(unsigned int x, unsigned int y,double size){
      x_=x;
      y_=y;
      size_=size;
   }


   virtual void sag_hallo(){
```

```
      std::cout<<"Ich bin ein OBJEKT und bin "<<size_<<" gross. Meine Position ist "<<x_<<"
   }


   protected:
   unsigned int x_,y_;
   double size_;


};



class wurfel: public objekt{
   public:
   wurfel(unsigned x, unsigned y, double size):
      objekt(x, y, size) {}
   void sag_hallo(){
      std::cout<<"Ich bin ein WÜRFEL und bin "<<size_<<" gross. Meine Position ist "<<x_<<"
   }

};

class kugel: public objekt{
   public:
   kugel(unsigned x, unsigned y, double size):
      objekt(x, y, size) {}
   void sag_hallo(){
      std::cout<<"Ich bin eine KUGEL und bin "<<size_<<" gross. Meine Position ist "<<x_<<"
   }

};

int main(){
   objekt* hello=new wurfel(3,4,5);
   hello->sag_hallo();
   hello=new kugel(7,8,9);
   hello->sag_hallo();
   return 0;
}
```