ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Algorithms: fundamentals

Prof. Dr. Goran Glavaš

26.10.2023

Content

- Building blocks of algorithms
- Algorithmic/programming paradigms
- Determinism in algorithms

Algorithm: Definitions

A process or set of rules to be followed in calculations or other **problem-solving operations**, especially by a computer.

Oxford dictionary

A finite sequence of rigorous instructions, typically used to **solve a class of specific problems** or to perform a computation.

Wikipedia

Any well-defined computational procedure that takes
some (set of) value(s) as input and produces some (set of) value(s) output.

Cormen et al.

In the beginning, there were only problems

- Algorithms are designed to solve **problems**
- Problems are commonly specified with:
  - Inputs
  - Desired outputs
  - *Optional*: Non-functional constraints
    - E.g., time or space complexity

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

In the beginning, there were problems

- *Optional*: Non-functional constraints
  - Refer to the constraints on the execution of the algorithm
  - **Time complexity**: duration of the algorithm execution
  - **Space complexity**: amount of computer memory needed for execution

Sorting Problem

Input: A sequence of *n* numbers $<a_1, a_2, \ldots, a_n>$

(Desired) Output: A permutation (reordering) of the input $<a'_1, a'_2, \ldots, a'_n>$ such that

$a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Time-complexity constraint: not more than n^2 elementary operations

What are algorithms built from?

- **Building blocks** of algorithms
  - Elementary operations
  - Sequential processing (**one** processing line)
  - Parallel processing (**multiple** processing lines)
  - Conditions (conditioned execution)
  - Loops (repetition)
  - Subprograms (modular construction of an algorithm)
  - Recursion (later in the course)

Let’s build an algorithm: **pseudocode**

Pseudocode is an artificial and informal language that helps us **develop algorithms**. It can be seen as a "text-based” tool for designing algorithms.

- Programming languages (e.g., Python, Java, C++) are artificial (as opposed to **natural** human languages), but formal

```
enroll_student_into_all_semester_courses
Input: stud_ID, sem_no
Data: studien_aufbau # maps semesters to courses
      wuecamp # maps courses to WueCampus courses/URLs

courses <- look into studien_aufbau for sem_no # our „step” 1
enrolled <- [] # empty list
for each course c in courses # step 2, iterating over all obtained courses
      wcc <- look into wuecamp for c # our „step” 2a
      success <- enroll(stud_ID, wcc)
      if success = True
         add wcc to enrolled

Output: enrolled
```

Let’s build an algorithm: **pseudocode**

Pseudocode is an artificial and informal language that helps us **develop algorithms**. It can be seen as a "text-based” tool for designing algorithms.

- **Building blocks** of algorithms
  -

```
enroll_student_into_all_semester_courses
Input: stud_ID, sem_no
Data: studien_aufbau # maps semesters to courses
      wuecamp # maps courses to WueCampus courses/URLs

courses <- look into studien_aufbau for sem_no # our „step” 1
enrolled <- [] # empty list
for each course c in courses # step 2, iterating over all obtained courses
      wcc <- look into wuecamp for c # our „step” 2a
      success <- enroll(stud_ID, wcc)
      if success = True
        add wcc to enrolled

Output: enrolled
```

Elementary (atomic) operations

- Cannot be broken into smaller suboperations
- Basiselemente eines Algorithmus, die nicht näher aufgeschlüsselt werden

Elementary operations

- Atomic elements of algorithms, not broken down further
  - Depends on how fine-grained the view we adopt is

An elementary operation is **one whose execution time is bounded by a constant for a particular machine and programming language**

- Types of elementary operations
  - Arithmetic operations (addition, subtraction, division, multiplication, ...)
  - Variable assignments
  - Value comparisons
  - Input/Output (that is, read/write operations)

Let’s build an algorithm: **pseudocode**

- **Building blocks** of algorithms
  -

```
enroll_student_into_all_semester_courses
Input: stud_ID, sem_no
Data: studien_aufbau # maps semesters to courses
      wuecamp # maps courses to WueCampus courses/URLs

courses <- look into studien_aufbau for sem_no # our „step” 1
enrolled <- [] # empty list
for each course c in courses # step 2, iterating over all obtained courses
      wcc <- look into wuecamp for c # our „step” 2a
      success <- enroll(stud_ID, wcc)
      if success = True
        add wcc to enrolled

Output: enrolled
```

read/write

comparison

assignment

Conditions

- Define whether one or more steps will be executed or not

- **Structure**:
  - `if [condition] then <step(s)> [else <other steps>]`

- **Nesting conditions**
  - Condition within a condition

```
if [cond1] then
  if [cond2] then
     <steps>
  else
     <steps>
else
  if [cond3] then
     <steps>
```

Conditions

- **Atomic conditions** are propositions of Boolean logic
  - They „return" a binary value: **true** or **false**
  - Typically value comparisons
    - *a = 4*?, *b* $\geq$ *c*?, *d in [17, 4, 3, 25]*?
  - Three standard boolean logic operations can be applied to conditions to create complex conditions
    - **Negation** („NOT" operator): **NOT [cond]**
      - True iff [cond] is False (and vice-versa)
    - **Conjuction** („AND" operator): **[cond1] AND [cond2]**
      - True iff both [cond1] and [cond2] are True
    - **Disjunction** („OR" operator): **[cond1] OR [cond2]**
      - True iff **at least** one of [cond1] and [cond2] are True

Loops (Schleife)

- **Repeated execution** of some steps
- Two types of loops
  - **WHILE**: the execution is repeated until the condition is satisfied
    - Number of repetitions not necessarily known in advance
  - **FOR**: the execution is repeated a fixed number of times

```
while [condition] do
  step1
  step2
  ...
```

```
for [iterator] do
  step1
  step2
  ...
```

- **[condition]** in **WHILE** loop: a Boolean expression as before
- **[iterator]** in **FOR** loop: generates a sequence of values over which to iterate

Loops (Schleife)

- Examples

```
x <- 10
while x > 5 do
  x <- x-1
print x
```

```
num <- [1, 2, 3]
sum <- 0
for x in num do
  sum <- sum + x*x
print sum
```

- Infinite loops (Endlosschleifen)
  - With **WHILE** loops, if the condition never becomes **False**

```
x <- 10
while x < 12 do
  x <- x-1
```

- Changing iterator while iterating
  - In **FOR** loops, some programming languages **don't allow** iterator to be changed inside of the loop
    - Python does allow it!
      - Behind the scene computes fixed list at first evaluation of the iterator

```
num <- [1, 2, 3]
sum <- 0
for x in num do
  sum <- sum + x*x
  num <- [4, 5, 6]
```

Functions, Subprograms, Modules

- Different problems sometimes share parts of the solution
  - Algorithms for those problems could thus have **common subparts**

- **Functions** (subprograms, modules)
  - Encapsulate the functionality that is needed across different algorithms
  - **Advantage**: avoid redundancy, implement shared functionality **once**, reuse wherever needed
  - **Advantage**: build algorithms compositionally, reusing existing solutions as much as possible (instead of from scratch for every new problem)

```
func enroll_student:
   Input: stud_id, wcc

   val_s <- check_valid_student(stud_id)
   val_c <- check_valid_course(wcc)

    if val_s = True AND val_c = True:
       add_student_course(stud_id, wcc)
       Output: True
    else:
       Output: False
```

Functions, Subprograms, Modules

- **Functions** (subprograms, modules)
  - Encapsulate the functionality that is needed across different algorithms
  - **Advantage**: avoid redundancy, implement shared functionality **once**, reuse wherever needed

```
func enroll_student:
  Input: stud_id, wcc
  val_s <- check_valid_student(stud_id)
  val_c <- check_valid_course(wcc)
   if val_s = True AND val_c = True:
      add_student_course(stud_id, wcc)
      Output: True
   else:
      Output: False
```

```
enroll_student_into_all_semester_courses
   Input: stud_ID, sem_no
          studien_aufbau # maps semesters to courses
    Data: wuecamp # maps courses to WueCampus courses/URLs

    courses <- look into studien_aufbau for sem_no # our „step" 1
    enrolled <- [] # empty list
    for each course c in courses # step 2, iterating over all obtained courses
          wcc <- look into wuecamp for c # our „step" 2a
          success <- enroll_student(stud_ID, wcc)
          if success = True
             add wcc to enrolled

    Output: enrolled
```

Functions (Subprograms, Modules)

- **Functions** (subprograms, modules)
  - **Advantage**: build algorithms compositionally, reusing as much existing solutions as possible (instead of from scratch for every new problem)

```
enroll_all_program_students_into_all_semester_courses
   Input: sem_no, studiengang
   Data: program_aufbaus # maps study programs to studienaufbau tables
         students_programs # maps programs to list of students enrolled into them

   studien_aufbau <- look into program_aufbaus for studiengang
   students <- look into students_programs for studiengang

   student_enrollments <- []
   for each student s in students:
        enrolled <- enroll_student_into_all_semester_courses(s, sem_no, studien_aufbau)
        add <s, enrolled> to student_enrollments

   Output: student_enrollments
```

Recursion

- Recursive algorithms solve **recursive problems**:
  - **Divisible** into subproblems of the same type as the original problem
  - **Solution to the subproblem** is part of the solution to the whole problem

Factorial (Fakultät) problem

Input: Natural number *n*
(Desired) Output: Factorial *n! = 1 * 2 * … * n*

Iterative solution

```
prod <- 1
for x in [2, 3, ..., n] do
  prod <- prod * x
```

But n! is a recursive problem

```
n! = n * (n-1)!
   = n * (n-1) * (n-2)!
   = ...
   = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

Recursion

- Recursive algorithms solve **recursive problems**:
  - **Divisible** into subproblems of the same type as the original problem
  - **Solution to the subproblem** is part of the solution to the whole problem

Factorial (Fakultät) problem

Input: Natural number *n*
(Desired) Output: Factorial $n! = 1 * 2 * ... * n$

Recursive solution (pseudocode)

```
func factorial
   Input: n
   if n = 1
     Output: 1
   else
     Output: n * factorial(n-1)
```

Recursive solution (Python)

```
def factorial(n):
  if n == 1:
     return 1
  else:
     return n * factorial(n-1)
```

Recursion

- Recursive algorithms have two main components:
  - Call-to-self: recursive call to the function within the function itself
  - Termination criterion of the recursion (*Abbruchkriterium der Rekursion*)
    - Without it, recursions would never end (infinite execution, similar to infinite loops)

Termination criterion

Recursive call

Recursive solution (pseudocode)

```
func factorial
   Input: n
   if n = 1
      Output: 1
   else
      Output: n * factorial(n-1)
```

Recursive solution (Python)

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

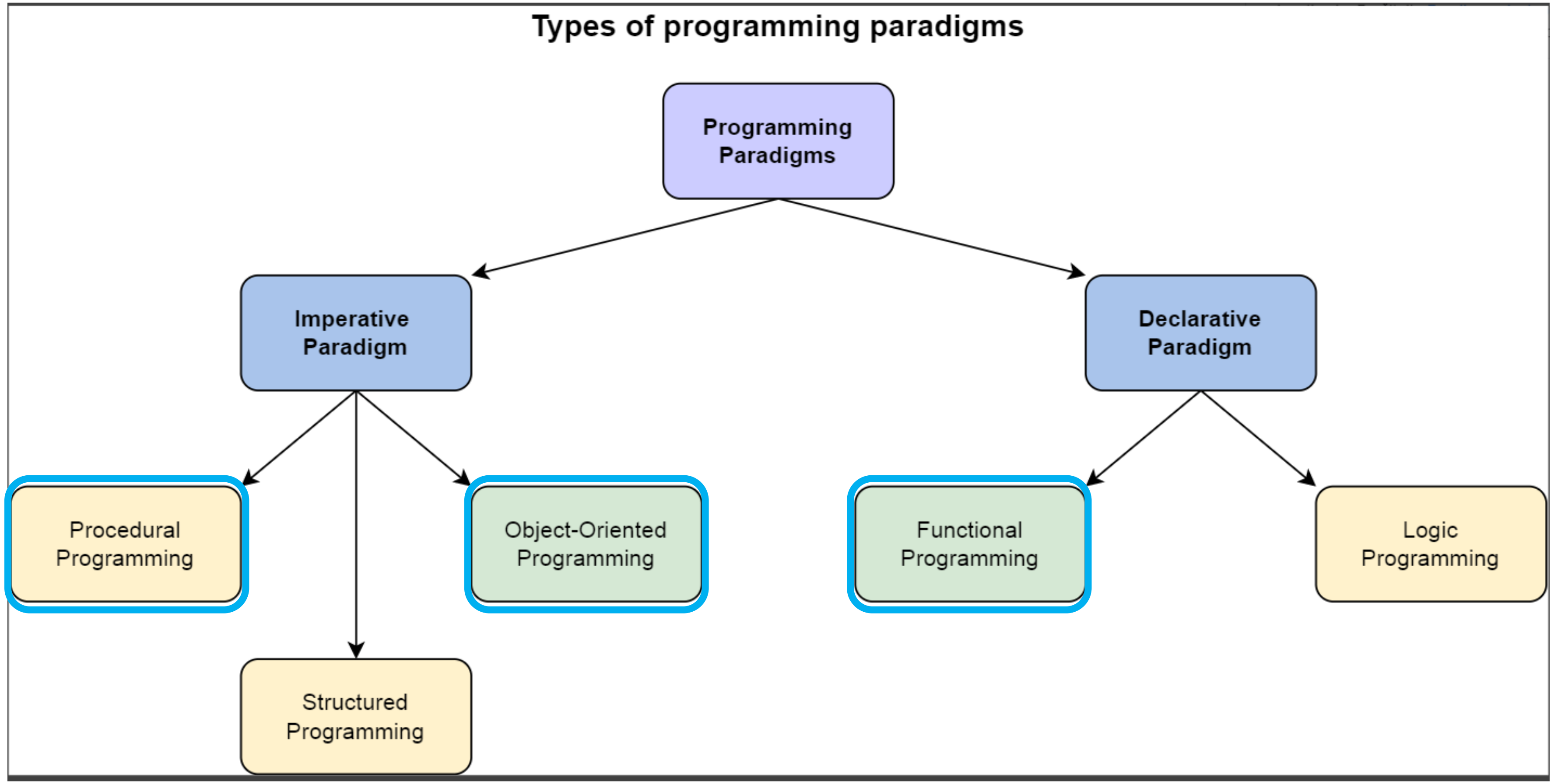
Content

- Building blocks of algorithms
- Algorithmic/programming paradigms
- Determinism in algorithms

Imperative vs. Declarative (and then objects...)

- Two main paradigms of algorithms:
  - **Imperative** (procedural) algorithms/programming
    - Explicitly describe steps to be executed (***how***)
    - Imperative programming languages: *C*, *C++*, *Java**, *C#**, *Python**, ...
  - **Declarative** (applicative) algorithms/programming
    - Describe what is to be done, but not (so directly) how (***what***)
    - Most well-known subtype is **functional programming**
      - Functional programming languages: *Haskell*, *Lisp*, *Clojure*, *Scala**, ...

Programming paradigms



Source: https://www.educative.io/blog/declarative-vs-imperative-programming

Imperative programming

Imperative programming is a paradigm that uses statements that change a **program's state**. An imperative program consists of commands for the computer to perform. It focuses on describing **how** a program operates step by step, rather than on high-level descriptions of its expected results.

Wikipedia

- The approach to algorithm design and programming that is:
  - The oldest, the most basic, the most intuitive, the most common/widespread
- Step-by-step execution (*control flow*) of commands – individual statements („elementary operations”) or function calls
  - With **conditions** and **loops** as basic building blocks
- **Stateful**: after every command, program/algorithm in **different state**;
  - If state is stored, program can continue running from it if interrupted

Types of imperative programming

- **Procedural programming**
  - Program/algorithm built from one or more **procedures** (aka subroutines or functions): these typically have well defined inputs and outputs
  - Much of AI/DS programming you'll write (e.g., in Python) will be procedural

- **Object-oriented programming**
  - Programming paradigm grounded in the notion of **objects**, which encapsulate **data** (*fields*) and **functions** (*methods*) that meaningfully go together
  - **Classes** define sets of data and functions, **objects** are instances of classes
    - In principle, it is possible to create any number of objects of some class
    - Classes determine the **type** of object (or **class = object type**)
  - Functions (or methods) of an object can access and modify the object's data
  - Even more of AI/DS programming you'll write (e.g., in Python) will be **OO**

Crash course in OOP

```
Class: Rectangle
    Data fields:
        height
        width
    Functions:
        create(h, w):
            height <- h
            width <- w
        surface():
            Output: height * width
        perimeter():
            Output: 2*(height + width)
```

Constructor!

```
Class: Circle
    Data fields:
        radius
    Functions:
        create(r):
            radius <- r
        surface():
            Output:
            radius*radius*pi
        perimeter():
            Output: 2*radius*pi
```

Core Principles of OOP

- **Encapsulation**
  - Data (fields) and methods that require that data **placed together**
  - Data and methods made *private* or *public*
  - Private fields (and methods) cannot be accessed/changed by object-external code, i.e., other objects
  - Public methods (and fields) are what the object exposes to „the world”
  - Python has classes/objects but *doesn’t have* private methods

Pseudocode

```
Class: Rectangle
    Data fields:
        height
        width
    Functions:
        create(h, w):
            height <- h
            width <- w
        surface():
            Output: height * width
        perimeter():
            Output: 2*(height + width)
```

Core Principles of OOP

- **Encapsulation**
  - Data (fields) and methods that require that data **placed together**
  - Data and methods made *private* or *public*
  - Private fields (and methods) cannot be accessed/changed by object-external code, i.e., other objects
  - Public methods (and fields) are what the object exposes to „the world”
  - Python has classes/objects but *doesn’t have* private methods

Python code

```
class Rectangle(object):
  def __init__(self, h, w):
    self.height = h
    self.width = w

  def surface(self):
    return self.height * self.width

  def perimeter(self):
    return 2*(self.height + self.width)
```

Core Principles of OOP

- **Abstraction**
  - Extension of encapsulation: complex programs contain (tens of) thousands of lines of codes
  - Encapsulation into classes provides a **layer of abstraction** – mental organization of code easier
  - Every object exposes only high-level mechanism for interacting with it (in a sense, *interface*)
    - Abstracts away low-level inner workings
  - If another object/code needs the ***surface*** of a Rectangle (or Circle), it just calls the method
    - Computation **abstracted away** from the calling object/code, it's the responsibility of the Rectangle class

Python code

```
class Rectangle(object):
  def __init__(self, h, w):
    self.height = h
    self.width = w

  def surface(self):
    return self.height * self.width

  def perimeter(self):
    return 2*(self.height + self.width)
```

Core Principles of OOP

- **Inheritance**
  - Complex programs often involve dealing with **similar objects** (objects of the same *„type"*)
  - Similar objects have **similar properties**
    - Share some code and logic but not exactly the same
    - Rectangle and Circle both geometric shapes with a measurable surface and perimeter
  - **Idea**: abstract away the „shared stuff" into something like a „parent class"
  - Parent class can typically have multiple children classes that inherit from it
    - In most OO programming languages, a class can have only one parent
  - **Abstract classes**: cannot be instantiated

Core Principles of OOP

- **Inheritance**: let's slightly modify our Rectangle and Circle classes

```
Class: Rectangle
    Data fields:
        color
        height
        width
    Functions:
        create(h, w, c):
            height <- h
            width <- w
            color <- c

        surface():
            Output: height * width
        perimeter():
            Output: 2*(height + width)
        change_color(c_new):
            color <- c_new
```

```
Class: Circle
    Data fields:
        color
        radius
    Functions:
        create(r, c):
            radius <- r
            color <- c

        surface():
            Output: radius*radius*pi
        perimeter():
            Output: 2*radius*pi
        change_color(c_new):
            color <- c_new
```

Core Principles of OOP

- **Inheritance**: a **parent class** GeoShape that Rectangle and Circle inherit from

```
Class: GeoShape
    Data fields:
      color
    Functions:
        create(c):
          color <- c

        surface():
          # cannot be implemented
        perimeter():
          # cannot be implemented
        change_color(c_new):
            color <- c_new
```

```
Class: Rectangle inh GeoShape
    Data fields:
        height
        width
    Functions:
        create(h, w, c):
            parent.create(c)
            height <- h
            width <- w
        surface():
            Output: height * width
        perimeter():
            Output: 2*(height + width)
```

```
Class: Circle inh GeoShape
    Data fields:
        radius
    Functions:
        create(r, c):
            parent.create(c)
            radius <- r
        surface():
            Output: height * width
        perimeter():
            Output: 2*(height + width)
```

Core Principles of OOP

- **Inheritance**: a **parent class** GeoShape that Rectangle and Circle inherit from

```
class GeoShape(object):
  def __init__(self, c):
    self.color = c

  def surface(self):
    raise NotImplementedError(„.")

  def perimeter(self):
    raise NotImplementedError(„.")

  def change_color(self, c_new):
    self.color = c_new
```

```
class Rectangle(GeoShape):
  def __init__(self, h, w, c):
    super().__init__(c)
    self.height = h
    self.width = w

  def surface(self):
    return self.height * self.width

  def perimeter(self):
    return 2*(self.height + self.width)
```

```
class Circle(GeoShape):
  def __init__(self, r, c):
    super().__init__(c)
    self.radius = r

  def surface(self):
    return self.radius**2 * math.pi

  def perimeter(self):
    return 2*self.radius*math.pi
```

```
rec1 = Rectangle(2, 5, "blue")
print(rec1.color)
rec1.change_color("red")
print(rec1.color)
```

```
circ1 = Circle(10, "green")
print(circ1.color)
circ1.change_color("yellow")
print(circ1.color)
```

Core Principles of OOP

- **Polymorphism**
  - Children may have **a different implementation of a method** (between them and) compared to the parent
  - But we can still call **the same function** for objects of different children classes (all classes that inherit the same parent class)

```
rec1 = Rectangle(3, 7, "violet")
rec2 = Rectangle(6, 2, "pink")
circ1 = Circle(6, "orange")
circ2 = Circle(4, "white")

shapes = [rec1, circ1, rec2, circ2]
for gs in shapes:
  print(gs.surface())
```

Declarative Programming

- We will focus on **functional programming**, the main branch of DP
  - Based on lambda calculus, a framework developed by Alonzo Church
  - Computation via (only) functions
- Everything is a **function**
  - Programming paradigm where we declare everything as calls to functions
  - „What to solve”, rather than „how to solve” (imperative)
- **Statelessness**
  - No (global) program state beyond the functions
    - No variables that store values, so that some other code can access them later
  - Information sharing exclusively though function inputs and outputs
  - No loops: iteration implemented via recursion!

Declarative Programming

- We will focus on **functional programming**, the main branch of DP
  - Based on lambda calculus, a framework developed by Alonzo Church
  - Computation via (only) functions
- **Expressions** instead of statements
  - Expressions are evaluated to produce a value
  - Statements are executed, i.e., assign values to variables (create **state**)
- **Pure functions** have two main properties:
  1. Always give the same value (output) for the same argument (input)
  2. **No side-effects**: do not modify inputs nor local/global variables
     - The values of variables can only be read (not (over)written): **referential transparency**
     - No state change (actually, no state at all): **statelessness**

Pros and Cons of Functional Programming

Pros	Cons
Pure functions are easier to understand, guaranteed not to change anything inadvertently	Writing pure functions can, in some cases, reduce the readability of code
functions as values (passed to other functions as parameters) often makes the code more readable and understandable	Writing programs in recursive style (instead of using loops) is counterintuitive for humans and requires mental adaptation (tough learning curves)
As variable values are immutable (cannot change value), debugging is easier	Writing pure functions is easy, but combining with (stateful) applications and I/O operations difficult
Pure functions are very suitable for concurrency/parallelism, as they don’t change states	Immutable values and recursion can result in slower execution (decrease in performance)
Lazy evaluation: values evaluated and stored only when really needed (avoids repeated evaluation)	

- **Multi-paradigm languages:** popular programming languages (Python, Java, C++, C#) are „generally imperative”, but support functional programming to some extent

Functional components (in Python)

- „Imperative” languages support **some degree** of functional programming
- Functions „first-class citizens”: anything you can do with „data” (variables), you can do with functions
  - Functions can be assigned to variables (same as data)

```
def func(x):
  print("Priting the provided input: " + str(x))

func(10)

# function can be assigned to a variable
some_other_variable = func

some_other_variable("will print this too")
```

Functional components (in Python)

- Anonymous functions:
  - Function without a name, defined **„on the fly”**, where you need it (instead of with „def”)
  - Typically for specific functions needed in a particular context that won’t be reused across contexts

- In Python: with „lambda” keyword

```
lambda <parameter_list>: <expression>
```

```
# anonymous functions
square = lambda x : x*x # or x**2
print(square(5))

numbers = [1, 2, 3, 4, 5]
for n in numbers:
  print(square(n))
```

Functional components (in Python)

- Anonymous functions:
  - Function without a name, defined **„on the fly”**, where you need it (instead of with „def”)
  - Typically for specific functions needed in a particular context that won’t be reused across contexts

- In Python: with „lambda” keyword

```
lambda <parameter_list>: <expression>
```

- Not necessary to assign a variable to a lambda expression before calling it

```
(lambda x1, x2, x3: x1 + x2 + x3)(9, 6, 6)
```

Functional components (in Python)

- **Map** and **Filter**
  - Built-in Python functions that fit the functional programming
  - Take another function as an argument
  - Designed to allow those who want to code **functionally** to **avoid looping**

```python
def square(x):
  return x**2

numbers = [1, 2, 3, 4, 5]

# iterator not a list, a function still
iterator = map(square, numbers)

# can ”evaluate” by casting to a list
squared_numbers = list(iterator)
```

- `map(<f>, <iterable>)`
  - <f> is a function to be applied on every element of <iterable> (list or array)

Functional components (in Python)

Functional

```python
def square(x):
  return x**2

numbers = [1, 2, 3, 4, 5]

# iterator not a list, a function still
iterator = map(square, numbers)

# can ”evaluate” by casting to a list
squared_numbers = list(iterator)
```

Imperative (Procedural)

```python
def square(x):
  return x**2

numbers = [1, 2, 3, 4, 5]

# instantiating empty list
squared_numbers = []

# iterating over numbers
for n in numbers:
  squared_numbers.append(square(n))
```

Functional components (in Python)

- **Map** and **Filter**
  - Built-in Python functions that fit the functional programming
  - Take another function as an argument
  - Designed to allow those who want to code **functionally** to **avoid looping**
- `filter(<f>, <iterable>)`
  - <f> is a boolean function, returns either True or False for each element of <iterable> (list or array)
- `map` and `filter` can be combined with anonymous functions (lambda)

```
def is_even(x):
  return x%2 == 0

numbers = [1, 2, 3, 4, 5]

# iterator not a list, a function still
iterator = filter(is_even, numbers)

# can "evaluate" by casting to a list
even_numbers = list(iterator)
```

```
num_minus_one = map(lambda x : x-1, numbers)
odd_numbers = filter(lambda x: x%2 == 1, numbers)
```

Content

- Building blocks of algorithms
- Algorithmic/programming paradigms
- Determinism in algorithms

Finite & deterministic algorithms

- In this course, we will (for the most part) deal with **finite** and **deterministic** algorithms

Finite algorithm

An algorithm is **finite** if it terminates in a finite numbers of steps for any given input

Deterministic algorithm

An algorithm that **returns a consistent result for any given input**. That means that if the algorithm is executed multiple times with the **same input**, it will produce the **same output** every time.

Non-deterministic algorithm

Algorithms that are **not guaranteed to return the same output for the same input**. In AI & DS, non-deterministic algorithms are most often **stochastic**, which means that the source of non-determinism is randomization (i.e., a random process).

Non-deterministic vs. Stochastic

- **Stochasticity** means there is randomness
- Every **stochastic algorithm** is non-deterministic but not every non-deterministic algorithm is necessarily stochastic
  - That is, there are other causes of non-determinism as well
  - Output needs to depend on something additional, not just input

Non-deterministic, not stochastic

```
def non_det_func(x):
  now = datetime.now()
  if now.second % 2 == 0:
    return x + 1
  else:
    return x + 2
```

Stochastic

```
import random

def stochastic_func(x):
  return x + random.randint(0, 5)
```

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟