# CODE STYLING AND CONVENTIONS

# Introduction

- Code styling and conventions are guidelines to the way a code should be written properly.
- The guidelines deal with the way the code is presented and its readability.

a + b

or

a+b ?

# Why do we need it?

- Most of the time spent working on software goes to maintenance
- A code is read much more often than it is written
- Hardly any piece of software is maintained only by the original author
- Code conventions improve the readability of the code, allowing authors to understand new code quicker

# Code Lay-out

To make the code readable and easy to understand quickly,
it is better to have a single Lay-out convention

Some features of the lay out for example are:

- Indentation – The use of 4 spaces per indentation level
  # not good indentation example:
  def abc():
  print('abc')
- Maximum Line Length – Limit all lines to a maximum of 79 characters

# Code Lay-out

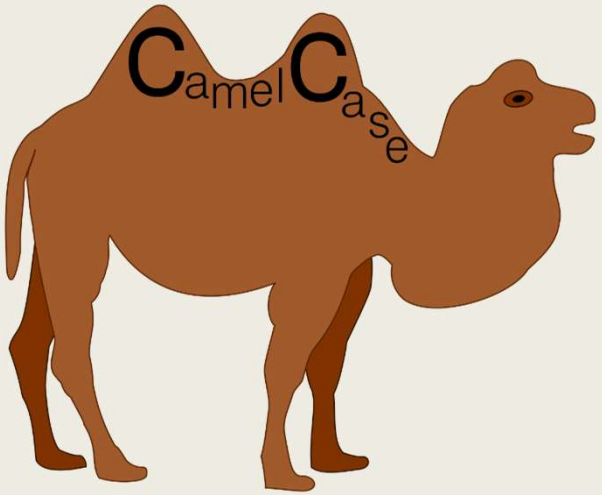
- Line breaking and indentation:

```
# good:
        foo = long_function_name(var_one, var_two,
                                 var_three, var_four)
# bad:
        foo = long_function_name(var_one, var_two,
            var_three, var_four)
```

# Naming Conventions

- When writing a code, we need to name our identifiers in a descriptive manner. This necessity becomes clear when writing a long code or when working with others.
- In addition, it is accustomed to have different naming styles for different uses.

# Naming Styles Examples




- a (single lowercase letter)
- B (single uppercase letter)
- lowercase (packages should normally use them)
- lower_case_with_underscores (Variables, functions, modules)
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (or CamelCase) (class names should normally use them)
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

# Names to Give and to Avoid

- You should name the identifiers in a meaningful way, even if it is longer
- Never use the characters ‘l’ (el) , ‘O’ , or ‘I’ (eye) , as they are indistinguishable from one and zero in some fonts
- When tempted to use ‘l’, use ‘L’ instead

# Comments

- Comments are very important to explain the code and to help remember what a segment of a code does
- Without comments things can get real confusing, real fast
- A program should contain comments that are multi-line and serve as documentation for others
- A program should contain comments that detail what a section of a code does

# Comment Examples

```
# This slide will be an example of comments in python. This comment is a
# documentation of the program: The program in this
# slide prints “example”

# This comment details what a segment of a code does:
# This function prints ’example’
def fun():
    print(‘example’)

fun()
```

# Magic Numbers

- Say we need to calculate a person's salary. They worked for 40 hours a week and earn 35 Shekels per hour:

```
salary = 40 * 35
```

- **What is wrong with this Code?**
- When we get back to this code after some time, we can't remember what each number indicates.
- One of these numbers might appear again:

```
[...]
bonus = 1.2 * 35
```

If our worker gets a raise next month to 40 Shekels per hour, we will want to change the code accordingly. If this number is hidden in all sorts of places in the code, we are more likely to miss it.

# Magic Numbers

- The right way:

```
weeklyHours = 40
hourlyPay = 35
salary = weeklyHours * hourlyPay
[...]
bonusPrecentage = 1.2
bonus = bonusPrecentage * hourlyPay
```

- Now we understand what these numbers mean
- If we want to change one of them, it is done in a single place in the code.

* in while and for loops it is acceptable that we still use 'i' and 'j' variables.

# No Copying and Pasting of Code

- Back to our salary calculator:

```
if worker.position == "manager":
    hourlyPay = 100
    weeklyHours = 45
    salary = hourlyPay * weeklyHours
elif worker.position == "secratary":
    hourlyPay = 50
    weeklyHours = 40
    salary = hourlyPay * weeklyHours
```

- Now, say that we change the way we calculate salaries next month. We will need to apply this change in several places - chances are we will miss one!

# No Copying and Pasting of Code

The right way:

```
if worker.position == "manager":
    hourlyPay = 100
    weeklyHours = 45
    salary = calcSalary(hourlyPay, weeklyHours)
elif worker.position == "secratary":
    hourlyPay = 50
    weeklyHours = 40
    salary = calcSalary(hourlyPay, weeklyHours)

def calcSalary(hourlyPay, weeklyHours):
    return hourlyPay * weeklyHours
```

# Splitting the Code into Functions

```
# extract velocity and x_0 from xData and tData to a linear graph and estimate x for
#  t =  tDest.
def predictX(xData, tData, sigX, sigT, tDest)
      dataPointsNum = len(x);
      if dataPointsNum != len(t) or dataPointsNum != len(sigX) or dataPointsNum !=
      len(sigT):
         print("Invalid data or error lists.")
         return
      if 0 in sigX  or 0 in sigT:
         print("Invalid errors, got sig = 0.")
         return
      fullSig = [];
      for i in range(dataPointsNum)
         fullSig.append((sigX(i)**2 + sigT(i)**2)**(1/2))
      term1 = sum([1/s**2 for s in fullSig]
      term2 =  sum([xData(i)**2./fullSig(i)**2 for i in range(dataPointsNum]);
      [...]
```

**The code is already too long for this slide, and we can't even understaned what is going on here!**

# Splitting the Code into Functions

```
# extract velocity and x_0 from xData and tData to a linear graph
and estimate x for
#  t =  tDest.
def predictX(xData, tData, sigX, sigT, tDest)
    dataPointsNum = len(x);
    if dataPointsNum != len(t) or dataPointsNum != len(sigX) or
    dataPointsNum != len(sigT):
        print("Invalid data or error lists.")
        return
    if 0 in sigX  or 0 in sigT:
        print("Invalid errors, got sig = 0.")
        return
    fullSig = [];
    for i in range(dataPointsNum)
        fullSig.append((sigX(i)**2 + (sigT(i)**2)**(1/2))
    term1 = sum([1/s**2 for s in fullSig]
    term2 =  sum([xData(i)**2./fullSig(i)**2 for i in
    range(dataPointsNum]);
    [...]
```

Check that the data is valid

Fit the data using chi squared minimization

# Splitting the Code into Functions

- The right way:

```
# extract velocity and x_0 from xData and tData to a linear
graph and estimate x for
#  t =  tDest.
def predictX(xData, tData, sigX, sigT, tDest)
   if not isValid(tData, xData,  sigT, sigX)
       return
   [x0, v] = fitLinear(tData, xData,  sigT, sigX)
   return x0 + v * tDest

# Check whether fitting data is valid for chi squared fit.
def isValid(xData, yData, sigX, sigY)
    [...]
# Perform a linear fit and return constant and slope
def fitLinear(xData, yData, sigX, sigY)
     [...]
```

# Exercise

- Try to understand what this code does, and style it:

```
def myFunc(d, m, y, d1)
    mLens = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = sum([mLens(i) for i in range(m - 1)])
    count = count + d
    if (y - 2008) % 4 == 0 and m > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - d1) % 7)
```

# Exercise

```
def myFunc(d, m, y, d1)
    mLens = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = sum([mLens(i) for i in range(m - 1)])
    count = count + d
    if (y - 2008) % 4 == 0 and m > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - d1) % 7)
```

# Exercise

```
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = day + month
    if (year - 2008) % 4 == 0 and month > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)
```

# Exercise

```
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = sum([monthLength(i) for i in range(month - 1)])
    count = count + day
    if (year - 2008) % 4 == 0 and month > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)
```

# Exercise

```
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = sum([monthLength(i) for i in range(month - 1)])
    count = count + day
    if isLeapYear(year) and month > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)

def isLeapYear(year)
    referenceYear = 2008
    return (year - referenceYear) % 4 == 0
```

# Exercise

```
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    count = sum([monthLength(i) for i in range(month - 1)])
    count = count + day
    if isLeapYear(year) and month > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)

def isLeapYear(year)
    referenceYear = 2008
    return (year - referenceYear) % 4 == 0
```

# Exercise

```
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    # We only count the days of the month before the current month
    count = sum([monthLength(i) for i in range(month - 1)])
    count = count + day
    if isLeapYear(year) and month > 2:
        count = count - 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)

def isLeapYear(year)
    referenceYear = 2008
    return (year - referenceYear) % 4 == 0
```

```
# Get date by day, month and year and calculate the weekday it will
# occure in. jan1Weekday represents the day of January 1st in that
# year *from 0 to 6* - Sunday is 0, Monday 1 etc.
def getWeekday(day, month, year, jan1Weekday)
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    # We only count the days of the month before the current month
    count = sum([monthLength(i) for i in range(month - 1)])
    count = count + day
    if isLeapYear(year) and month > 2:
        count = count + 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
        “Thursday”, “Friday”, “Saturday”]
return days((count - jan1Weekday) % 7)

# Check if the year is a leap year (has Feb 29th)
def isLeapYear(year)
    # 2008 was a leap year
    referenceYear = 2008
    return (year - referenceYear) % 4 == 0
```

```
# Get date by day, month and year and calculate the weekday it will
# occure in. jan1Weekday represents the day of January 1st in that
# year *from 0 to 6* - Sunday is 0, Monday 1 etc.
def getWeekday(day, month, year, jan1Weekday):
    monthLengths = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    # We only count the days of the month before the current month
    count = sum([monthLengths[i] for i in range(month - 1)])
    count = count + day
    if isLeapYear(year) and month > 2:
        count = count + 1;
    days = [“Sunday”, “Monday”,  “Tuesday”, “Wednesday”,
            “Thursday”, “Friday”, “Saturday”]
    return days[(count + jan1Weekday - 1) % 7)]

# Check if the year is a leap year (has Feb 29th)
def isLeapYear(year):
    # 2008 was a leap year
    referenceYear = 2008
    return (year - referenceYear) % 4 == 0
```

# Important things to remember

- Make our code as readable as possible for future users
- Our goal is that the code we program would be user-friendly
- The names we choose should be meaningful and not random
- Adding comments is crucial in order to optimize programmers efficiency