Control Flow and reserved words

```
          #      Control Flow is about how to change the sequence of instructions

          #      Following is a list of all reserved names in Python. These names can not be used
          #      for self-defined objects
CF001 >>>        """ False      class      finally    is         return
      ...            None       continue   for        lambda     try
      ...            True       def        from       nonlocal   while
      ...            and        del        global     not        with
      ...            as         elif       if         or         yield
      ...            assert     else       import     pass
      ...            break      except     in         raise      """
          #      most of these reserved names are part of some control flow syntax
          #      The following pages will show, how the these keywords are used
```

'if' - Conditional Execution

```
CF002 >>>    boolex = True # this can be any conditional expression or boolean value
CF003 >>>    a = 'nothing'

CF004 >>>    if not boolex:
      ...        a = 'yes'   # block of statements
CF005 >>>    a    # statement was not executed
      ==>    'nothing'

CF006 >>>    if boolex:
      ...        a = 'yes'   # block of statements
CF007 >>>    a    # statement was executed
      ==>    'yes'

        #    there is an 'else' branch
CF008 >>>    if not boolex:
      ...        a = 'yes'   # block of statements
      ...    else:
      ...        a = 'no'    # alternative block of statements
CF009 >>>    a    # the else part was executed
      ==>    'no'

        #    the keywords 'if' and 'else' also occur in a 'conditional assignment'
        #    a conditional assignement allows a shortcut for the above if...else

CF010 >>>    a = 'yes' if boolex else 'no'
        #    this form of a statement is new in Python and it was introduced
        #    on a demand to have something like the C/C++ statement:
CF011 >>>    "a = boolex ? 'yes' : 'no';"
```

'if' with many conditions

```
CF012 >>>   small, medium, big = 3, 7, 15
CF013 >>>   item = 8
CF014 >>>   if item <= small:
      ...      print("small")
      ...   elif item <= medium:
      ...      print("medium")
      ...   elif item <= big:
      ...      print("big")
      ...   else:
      ...      print("huge")
      p()   big

        #   Nested 'if' statements
CF015 >>>   if True:
      ...       if True:
      ...           if False:
      ...               pass
      ...           else:
      ...               pass
      ...   else:
      ...       pass
```

'for ... in' - Process objects in a sequence

```
          #     The for statement is used to access all elements in a list
CF016 >>>     fruit = ['apple', 'pear', 'banana', 'orange', 'kiwi', 'strawberry']
CF017 >>>     for element in fruit:
        ...          print(element)
        p()     apple
        p()     pear
        p()     banana
        p()     orange
        p()     kiwi
        p()     strawberry

          #     check all elements and skip some - use 'continue'
CF018 >>>     for fr in fruit:
        ...          if len(fr) > 5:
        ...              continue    # returns to the begin of the loop, and fetches the next element
        ...          print("selected fruit: {}".format(fr.upper()))
        p()     selected fruit: APPLE
        p()     selected fruit: PEAR
        p()     selected fruit: KIWI
CF019 >>>     fr   # the last element is still there
        ==>     'strawberry'
```

Termination of a 'for' loop

```
      #    Terminate a 'for' loop at some condition (find something with 'z')
      #    the 'else' part of a loop is executed, when the iteration reaches its end

CF020 >>>  for fr in fruit:
      ...      if fr[0] == 'z':
      ...          print("element, which terminates the loop:", fr)
      ...          break
      ...      print("found:", fr)
      ...  else:
      ...      print("nothing found with 'z'")
      p()  found: apple
      p()  found: pear
      p()  found: banana
      p()  found: orange
      p()  found: kiwi
      p()  found: strawberry
      p()  nothing found with 'z'
```

Termination of a 'for' loop

```
#       Terminate a 'for' loop at some condition (find something with 'b')
#       the 'else' part is skipped, when the loop is terminated by a 'break'

CF021 >>>   for fr in fruit:
      ...       if fr[0] == 'b':
      ...           print("element, which terminates the loop:", fr)
      ...           break
      ...       print("found:", fr)
      ...   else:
      ...       print("loop was exhausted")
      p()   found: apple
      p()   found: pear
      p()   element, which terminates the loop: banana
```

'for ... in' - just for counting

```
       #     'for' works with sequences. A sequence often is numbers from 'range()'
CF022 >>>    for num in range(3):
      ...        print("in the range:", num)
      p()    in the range: 0
      p()    in the range: 1
      p()    in the range: 2

       #     look at this:
CF023 >>>    range(3)       # this is an object, which 'generates' numbers
      ==>    range(0, 3)
CF024 >>>    list(range(3))     # This makes a list, consisting of the generated numbers
      ==>    [0, 1, 2]

       #     its good to understand the range() function
CF025 >>>    list(range(4,8))     # from 4 up to, but not including 8
      ==>    [4, 5, 6, 7]
CF026 >>>    list(range(0,10,2))    # counting in steps of 2
      ==>    [0, 2, 4, 6, 8]
CF027 >>>    list(range(13,-11,-5))     # counting can be backwards and negative
      ==>    [13, 8, 3, -2, -7]
```

while True ... for external events

```
       #     sometimes we need a loop, but we can not know before, when it ends

CF028 >>>    from external import get_data

CF029 >>>    dlist = []
CF030 >>>    while True:
      ...        data = get_data()
      ...        if data is None:
      ...            break
      ...        dlist.append(data)

CF031 >>>    print("{} data elements were received".format(len(dlist)))
      p()    9 data elements were received

       #     An 'else' statement would never be reached from a 'while True' loop
```

while [condition] ... working on data

```
CF032 >>>   text = "you can't blame gravity for falling in love"  # a quote from A. Einstein

CF033 >>>   while text:     # which is the same as while len(text) > 0
      ...       first_ch = text[0]
      ...       mylist = text.split(first_ch)  # yes, some string processing here
      ...       text = ''.join(mylist)   # ... and here
      ...       print('"{}" is found {} times, rest: {}'
      ...             .format(first_ch, len(mylist)-1, text))
      p()   "y" is found 2 times, rest: ou can't blame gravit for falling in love
      p()   "o" is found 3 times, rest: u can't blame gravit fr falling in lve
      p()   "u" is found 1 times, rest:  can't blame gravit fr falling in lve
      p()   " " is found 7 times, rest: can'tblamegravitfrfallinginlve
      p()   "c" is found 1 times, rest: an'tblamegravitfrfallinginlve
      p()   "a" is found 4 times, rest: n'tblmegrvitfrfllinginlve
      p()   "n" is found 3 times, rest: 'tblmegrvitfrflligilve
      p()   "'" is found 1 times, rest: tblmegrvitfrflligilve
      p()   "t" is found 2 times, rest: blmegrvifrflligilve
      p()   "b" is found 1 times, rest: lmegrvifrflligilve
      p()   "l" is found 4 times, rest: megrvifrfigive
      p()   "m" is found 1 times, rest: egrvifrfigive
      p()   "e" is found 2 times, rest: grvifrfigiv
      p()   "g" is found 2 times, rest: rvifrfiiv
      p()   "r" is found 2 times, rest: viffiiv
      p()   "v" is found 2 times, rest: iffii
      p()   "i" is found 3 times, rest: ff
      p()   "f" is found 2 times, rest:
```

Exceptions - what can go wrong, will go wrong

```
#       some situations arise completely unexpected
#       others can be expected, but we cannot know if or when they will occur

CF034 >>>   from external import get_capitals   # some external function

CF035 >>>   countries = 'italy belgium germany portugal sweden '\
      ...               'finland hungary bulgaria'.split()

CF036 >>>   tab_format = "{:15s} {}"
CF037 >>>   print(tab_format.format("country", "capital"))
      ...   print(tab_format.format("-"*10, "-"*10))
      ...   for country in countries:
      ...       capital = get_capitals(country)
      ...       print(tab_format.format(country, capital))
      p()   country         capital
      p()   ----------      ----------
      p()   italy           rome
      p()   belgium         brussels
      p()   germany         berlin
      p()   portugal        lisbon
      err!  KeyError('sweden',)
```

Exceptions - we can handle exceptions

```
CF038 >>>   unknown = []
CF039 >>>   print(tab_format.format("country", "capital"))
      ...   print(tab_format.format("-"*10, "-"*10))
      ...   for country in countries:
      ...       try:
      ...           capital = get_capitals(country)
      ...       except KeyError:
      ...           unknown.append(country)
      ...       else:
      ...           print(tab_format.format(country, capital))
      ...   print('capital unknown for:', unknown)
      p()   country         capital
      p()   ----------      ----------
      p()   italy           rome
      p()   belgium         brussels
      p()   germany         berlin
      p()   portugal        lisbon
      p()   finland         helsinki
      p()   bulgaria        sofia
      p()   capital unknown for: ['sweden', 'hungary']
```

Not interested in specific errors

```
      #     when resource (like a file) is accessed (reserved)
      #     it should be freed, even if some (unknown, nospecific) error occurs

CF040 >>>   from external import failing_function as func

CF041 >>>   def process_line(line):
      ...       response = func(line)
      ...       print("data: {}, response: {}".format(line, response))

CF042 >>>   infile = open('testfile01.txt', mode='r')  # open files should be closed
CF043 >>>   try:
      ...       for line in infile:
      ...           process_line(line.strip())
      ...   finally:    # the finally part is always executed
      ...      print("infile is closed")
      ...      infile.close()     # closing of the file is assured
      p()   data: line number   0, response: ok
      p()   data: line number   1, response: ok
      p()   data: line number   2, response: ok
      p()   infile is closed
      err!  ExoticError('There was something strange',)
```

No solution for an error, but some helpful action

```
#      sometimes its important to catch errors, but not all errors can be handeled
#      If errors can not be handeled, they must be 're-raised'

#      Its a deadly sin, to catch an error and ignore it silently!

CF044 >>>   infile = open('testfile01.txt', mode='r')
CF045 >>>   try:
      ...       for line in infile:
      ...           try:
      ...                process_line(line.strip())
      ...           except KeyError:
      ...               pass   # assumed that a KeyError can be safely ignored
      ...           except (IndexError, ValueError) as excp: # catch multiple exceptions
      ...               print("ecountered an expected error", excp)  # print, but continue
      ...           except Exception:  # This catches all other errors!
      ...               print("unexpected error at line: '{}'".format(line.strip()))
      ...               raise   # this re-raises the error, so can be handled somewhere else
      ...               # it is very important, to re-raise after catching 'Exception'
      ...   finally:
      ...      infile.close()
      p()   data: line number   0, response: ok
      p()   data: line number   1, response: ok
      p()   data: line number   2, response: ok
      p()   unexpected error at line: 'line number   3'
      err!  ExoticError('There was something strange',)
```

more to come ...

```
#    def ... return
#    global
#    raise
#
#    class
#    del
#    True, False, None
#    and, or, not

#    Imperative statements are statements, which perform basic of the language
```

Imperative Statements

# 'import'