OOTS Command Evaluation

```
#       At the prompt '>>>' any valid expression or statement can be entered
#       The Python shell shows the results of the evaluation or
#       executes the statement

#       The hash '#' starts a comment
```

--- Simple values ---

```
B001 >>>    5     # number (integer)
     ==>    5
B002 >>>    234923059834209485209680294865203463426982034     # integers may be long!
     ==>    234923059834209485209680294865203463426982034
B003 >>>    3.1417    # decimal number
     ==>    3.1417
B004 >>>    'some text'    #  a string: text in apostrophes
     ==>    'some text'
B005 >>>    "  more text "    #  another string:  text in quotes (space is preserved)
     ==>    '  more text '
```

--- More about strings ---

```
      #     for strings the output from evaluation is different from print() output
B006 >>>    "this is John's book"     # embed an apostrophe in a quoted string
     ==>    "this is John's book"
B007 >>>    print("this is John's book")     # print() shows the data without additional quotes
     p()    this is John's book
B008 >>>    'embed "qotes" in a string'    # the other way round
     ==>    'embed "qotes" in a string'
B009 >>>    print('embed "qotes" in a string')
     p()    embed "qotes" in a string
B010 >>>    len('abcde')    # get the length of a string
     ==>    5
```

--- Escaping ---

```
      #     some characters can or must be be 'escaped' with a backslash '\'
      #     escaping allows to enter special characters into strings
      #     the 'effect' of the special characters is only visible with print()
B011 >>>    "escape a \" and a \'"     # escaping for the " is necessary
     ==>    'escape a " and a \''
B012 >>>    print("escape a \" and a \'")     # the print output looks slightly different
     p()    escape a " and a '
B013 >>>    'escape a \" and a \''     # escaping for the ' is necessary
     ==>    'escape a " and a \''
B014 >>>    print('escape a \" and a \'')     # different output again
     p()    escape a " and a '
      #     its also possible to insert a "newline" with '\n'
B015 >>>    "this text is split\ninto two lines"  # evaluation leaves the \n untouched
     ==>    'this text is split\ninto two lines'
B016 >>>    print("this text is split\ninto two lines")  # print() resolves the \n
     p()    this text is split
     p()    into two lines
```

--- Triple quoted strings ---

```
B017 >>>    '''string surrounded with triple apostrophes'''    # used for what?
     ==>    'string surrounded with triple apostrophes'
B018 >>>    """can contain "this" and 'that'"""  # quotes and apostrophes without escape
     ==>    'can contain "this" and \'that\''
       #    its also possible to span several lines
B019 >>>    '''A string starts here
     ...       is continued
     ...    and ends here.'''  # again, evaluation does not 'resolve' the newlines
     ==>    'A string starts here\n   is continued\nand ends here.'
B020 >>>    print('''A string starts here,
     ...       is continued
     ...    and ends here.''')  # but print() does, leading spaces are preserved
     p()    A string starts here,
     p()       is continued
     p()    and ends here.
B021 >>>    "a\tb"  # this is a 'tabulator'
     ==>    'a\tb'
B022 >>>    print("a\tb") # the "effect" of a "tabulator" depends on the output device
     p()    a■b
```

--- Mathematical expressions ---

```
B023 >>>    2+17
     ==>    19
B024 >>>    2+17*3       # priority rules matter
     ==>    53
B025 >>>    (2+17)*3     # parenthesis work
     ==>    57
B026 >>>    3,5 + 2     # attention!
     ==>    (3, 7)
       #    a comma (,) actually forms a tuple (sequence of values)
B027 >>>    3.5 + 2     # try again ...
     ==>    5.5
```

--- More Math ---

```
B028 >>>    7/0    # not every operation is valid
     err!   ZeroDivisionError('division by zero',)
B029 >>>    7/1    # for a decimal or a division, the result is a decimal
     ==>    7.0
B030 >>>    1+2+3+4+5+6+7+9
     ==>    37
B031 >>>    1+2+3+4+5+6+7+9+0.0
     ==>    37.0
B032 >>>    3**4    # 3 to the power of 4
     ==>    81
B033 >>>    pow(3, 4)   # same result with a builtin function
     ==>    81
```

--- More Math ---

```
B034 >>>    -3    # one-sided minus
     ==>    -3
B035 >>>    7-3    # two-sided minus
     ==>    4
B036 >>>    7-(-3)
     ==>    10

B037 >>>    13/4
     ==>    3.25
B038 >>>    13//4   # integer divison - ignores the rest
     ==>    3
B039 >>>    13%4    # this gives the missing rest
     ==>    1
```

--- Numeric Conversions ---

```
B040 >>>    abs( 7.3)  # absolut value - ignore a negative sign
     ==>    7.3
B041 >>>    abs(-7.3)
     ==>    7.3
B042 >>>    int( 7.3)    # returns a integer by ignoring the decimal part
     ==>    7
B043 >>>    int(-7.3)
     ==>    -7
B044 >>>    int("4")    # convert a string
     ==>    4
B045 >>>    int("   -4  ")    # leading and trailing spaces are ok
     ==>    -4
B046 >>>    int("   - 3")
     err!   ValueError("invalid literal for int() with base 10: '   - 3'",)
B047 >>>    int("-4.2")    # and it must be an integer value
     err!   ValueError("invalid literal for int() with base 10: '-4.2'",)
B048 >>>    float(3)
     ==>    3.0
B049 >>>    float(3.9)
     ==>    3.9
B050 >>>    float(" -17.33")    # convert a string
     ==>    -17.33
```

--- Floating point math has its limits ---

```
B051 >>>    10/3    # precision is finite
     ==>    3.3333333333333335
B052 >>>    round(1.3)   # round down
     ==>    1
B053 >>>    round(1.6)   # round up
     ==>    2
B054 >>>    round(1.5)   # up or down?
     ==>    2
B055 >>>    round(2.5)   # up or down?
     ==>    2
B056 >>>    0.1 + 0.2
     ==>    0.30000000000000004
B057 >>>    0.1 + 0.2 == 0.3   # should be True
     ==>    False
       #    for more details read the Python Tuorial: "Floating Point Arithmetic"
```

--- Boolean expressions ---

```
B058 >>>    True     # a special name, considered as logical '1'
     ==>    True
B059 >>>    False    # considered as logical '0'
     ==>    False
B060 >>>    2 == 3   # comparison operator: equal
     ==>    False
B061 >>>    2 != 3   # not equal
     ==>    True
B062 >>>    2 < 3
     ==>    True
B063 >>>    True and True     # 'and' is considered as logical multiplication
     ==>    True
B064 >>>    True and False
     ==>    False
B065 >>>    True or False     # 'or' is considered as logical addition
     ==>    True
B066 >>>    not True          # 'not' is a logical minus
     ==>    False
B067 >>>    not False
     ==>    True
B068 >>>    not True or True    # attention
     ==>    True
B069 >>>    not (True or True)  # 'not' has precedence over 'and' has preedence over 'or'
     ==>    False
```

--- Boolean expressions ---

```
B070 >>>    3 > 2 and 3 < 4
     ==>    True
B071 >>>    2 < 3 < 4
     ==>    True
B072 >>>    bool()    # convert 'something' into a boolean value
     ==>    False
B073 >>>    bool(False)
     ==>    False
B074 >>>    bool(0)
     ==>    False
B075 >>>    bool(13)     # anything != 0 should be True
     ==>    True
B076 >>>    bool('')
     ==>    False
B077 >>>    bool(' ')    # anything not "empty" should be True
     ==>    True
B078 >>>    bool(None)   # None represents 'nothing', 'nul'
     ==>    False
```

--- Variables and Assignments ---

```
B079 >>>    a = 1
B080 >>>    a1 = 3
B081 >>>    A = 5       # case (upper/lower) matters, A != a
B082 >>>    A_3_ = 7
B083 >>>    a
     ==>    1
B084 >>>    a1, A, A_3_     # comma separated values form a 'tuple'
     ==>    (3, 5, 7)
B085 >>>    atuple = a, a1, A_3_     # assign multiple values to a tuple
B086 >>>    atuple
     ==>    (1, 3, 7)
B087 >>>    print("tuple:", atuple)
     p()    tuple: (1, 3, 7)
B088 >>>    print("single values:", a, a1, A_3_)
     p()    single values: 1 3 7
B089 >>>    b = c = a     # multiple assignments
B090 >>>    a, b, c
     ==>    (1, 1, 1)
B091 >>>    del b, A    # del removes a name from the name table
B092 >>>    a, b, c
     err!   NameError("name 'b' is not defined",)
```

--- Variables and Assignments ---

```
B093 >>>    under_scored = 'hello'   # name convention for variables and functions
B094 >>>    camelCase = 'world'      # name convention for classes (will come later)
B095 >>>    x, y, z = atuple    # unpack a tuple
B096 >>>    print(x, y, z)
     p()    1 3 7
B097 >>>    x == a
     ==>    True
B098 >>>    y == a1, z == A_3_
     ==>    (True, True)
```

Special assignment

```
B099 >>>    a
     ==>    1
B100 >>>    a += 9    # same as a = a+9
B101 >>>    a -= 4
B102 >>>    a *= 4
B103 >>>    a
     ==>    24
```

--- Tuples ---

```
B104 >>>    ftup = 'apple', 'pear', 'cherry', 'plum'
B105 >>>    ftup
     ==>    ('apple', 'pear', 'cherry', 'plum')
B106 >>>    gtup = ('apple', 'pear', 'cherry', 'plum')
B107 >>>    gtup
     ==>    ('apple', 'pear', 'cherry', 'plum')
B108 >>>    ftup == gtup
     ==>    True
B109 >>>    ftup[0]
     ==>    'apple'
B110 >>>    ftup[1]
     ==>    'pear'
B111 >>>    gtup[2] = 'banana'
     err!   TypeError("'tuple' object does not support item assignment",)
B112 >>>    gtup
     ==>    ('apple', 'pear', 'cherry', 'plum')
```

--- Lists ---

```
B113 >>>    hlist = []    # empty list
B114 >>>    hlist = [3.14159, '', 2+7+5, None, 'nobody', 0, False]
B115 >>>    hlist
     ==>    [3.14159, '', 14, None, 'nobody', 0, False]
B116 >>>    len(hlist)
     ==>    7
B117 >>>    hlist[4]
     ==>    'nobody'
B118 >>>    hlist[3] = 777
B119 >>>    hlist
     ==>    [3.14159, '', 14, 777, 'nobody', 0, False]
B120 >>>    tuple(hlist)   # copy and convert
     ==>    (3.14159, '', 14, 777, 'nobody', 0, False)
B121 >>>    list(gtup)     # copy and convert
     ==>    ['apple', 'pear', 'cherry', 'plum']
```

--- Lists & Tuples: Indexing and Slicing ---

```
B122 >>>    alist = ['Tom', 'Jude', 'Mary', 'Jack', 'Rose', 'Rob']
B123 >>>    alist[0], alist[5]
     ==>    ('Tom', 'Rob')
B124 >>>    alist[6]
     err!   IndexError('list index out of range',)
B125 >>>    alist[-1]    # last element
     ==>    'Rob'
B126 >>>    alist[-3]    # third from the end
     ==>    'Jack'
B127 >>>    alist[0:2]   # part (slice) of the list, including 0 but not 2
     ==>    ['Tom', 'Jude']
B128 >>>    alist[1:4]
     ==>    ['Jude', 'Mary', 'Jack']
B129 >>>    alist[3:]    # 3 and all following
     ==>    ['Jack', 'Rose', 'Rob']
B130 >>>    alist[3:9]   # if the list is not long enough, just return what is there
     ==>    ['Jack', 'Rose', 'Rob']
B131 >>>    alist[:3]    # all up to, but not including 3
     ==>    ['Tom', 'Jude', 'Mary']
B132 >>>    alist[-2:]   # the last 2 elements
     ==>    ['Rose', 'Rob']
```

Lists can be modified

```
B133 >>>    alist
     ==>    ['Tom', 'Jude', 'Mary', 'Jack', 'Rose', 'Rob']
B134 >>>    alist[2] = 'Joan' # replace one element
B135 >>>    del alist[-2:]    # delete the last 2 elements
B136 >>>    alist.append('Peter')   # append a single value
B137 >>>    alist
     ==>    ['Tom', 'Jude', 'Joan', 'Jack', 'Peter']

B138 >>>    more_names = ['Mark', 'Dora', 'Liv']
B139 >>>    alist.extend(more_names)   # append a list
B140 >>>    alist
     ==>    ['Tom', 'Jude', 'Joan', 'Jack', 'Peter', 'Mark', 'Dora', 'Liv']
B141 >>>    alist[3:3] = ['Bert']    # insert a list
B142 >>>    alist.insert(5, 'Susan') # insert a single value
B143 >>>    alist
     ==>    ['Tom', 'Jude', 'Joan', 'Bert', 'Jack', 'Susan', 'Peter', 'Mark', 'Dora',
B144 >>>    sorted(alist)   # this returns a copy
     ==>    ['Bert', 'Dora', 'Jack', 'Joan', 'Jude', 'Liv', 'Mark', 'Peter', 'Susan',
B145 >>>    alist
     ==>    ['Tom', 'Jude', 'Joan', 'Bert', 'Jack', 'Susan', 'Peter', 'Mark', 'Dora',
B146 >>>    alist.sort()    # this modifies the original list
B147 >>>    alist
     ==>    ['Bert', 'Dora', 'Jack', 'Joan', 'Jude', 'Liv', 'Mark', 'Peter', 'Susan',
```

--- More on Strings ---

```
      #     Strings are like a tuple of single characters
B148 >>>    xstr = "Hello World"
B149 >>>    len(xstr)
     ==>    11
B150 >>>    xstr[0]
     ==>    'H'
B151 >>>    xstr[-1]
     ==>    'd'
B152 >>>    xstr[-3:]
     ==>    'rld'
B153 >>>    xstr[3:8]
     ==>    'lo Wo'
```

--- String methods ---

```
      #     there are many methods, that work with strings.
B154 >>>    xstr.lower()     # the original string is never modified
     ==>    'hello world'
B155 >>>    xstr.upper()     # string methods always return copies
     ==>    'HELLO WORLD'
B156 >>>    xstr.center(20, '-')
     ==>    '----Hello World-----'
B157 >>>    xstr.endswith('World')    # return a boolean value
     ==>    True
B158 >>>    xstr.endswith('world')    # return a boolean value, case matters
     ==>    False
B159 >>>    xstr.find('l')      # search a string inside
     ==>    2
B160 >>>    xstr.find('l', 2)  # search from a starting position
     ==>    2
B161 >>>    xstr.find('l', 3)  # search from a starting position
     ==>    3
B162 >>>    xstr.find('l', 5)  # search from a starting position
     ==>    9
```

--- More about Strings ---

```
B163 >>>    noisy = '   some text \n'
B164 >>>    noisy.strip()     # strip() removes 'whitespace'
     ==>    'some text'
B165 >>>    noisy.rstrip()
     ==>    '   some text'
B166 >>>    noisy.lstrip()
     ==>    'some text \n'
B167 >>>    noisy.split()     # split() ignores whitespace
     ==>    ['some', 'text']
B168 >>>    noisy.split('e')    # split returns a list
     ==>    ['   som', ' t', 'xt \n']
B169 >>>    noisy.split(' ')
     ==>    ['', '', '', 'some', 'text', '\n']
B170 >>>    noisy.split('$')    # split always returns a list with at least one element
     ==>    ['   some text \n']
```

--- Printing and Formatting Text ---

```
B171 >>>    integ = -234
B172 >>>    hestr = 'Hello'
B173 >>>    wostr = 'World'
B174 >>>    listr = 'Lisbon'

B175 >>>    print(integ)
     p()    -234
B176 >>>    print(hestr, listr)
     p()    Hello Lisbon
       #    put more than one instruction on one line, possible but not recommended
B177 >>>    print(hestr); print(wostr)     # print two strings side by side?
     p()    Hello
     p()    World
B178 >>>    print(hestr, end=''); print(wostr)    # if 'end' is not specified, it defaults to '\n'
     p()    HelloWorld
B179 >>>    print(hestr, end=' '); print(wostr)
     p()    Hello World
```

--- The format() string methond

```
B180 >>>    'Greetings: {} {}, {} {}'.format(hestr, listr, hestr, wostr)
     ==>    'Greetings: Hello Lisbon, Hello World'
B181 >>>    'Greetings: {0} {1}, {0} {2}'.format(hestr, listr, wostr)
     ==>    'Greetings: Hello Lisbon, Hello World'
B182 >>>    'Greetings: {0:>10} {1:<10}, {0} {2}'.format(hestr, listr, wostr)
     ==>    'Greetings:      Hello Lisbon    , Hello World'
B183 >>>    'Greetings: {0:>4} {1:<4}, {0} {2}'.format(hestr, listr, wostr)  # no truncatio
     ==>    'Greetings: Hello Lisbon, Hello World'

B184 >>>    'Say {}, {} times'.format(hestr, 7)    # format numbers
     ==>    'Say Hello, 7 times'
B185 >>>    'Say {}, {:+4d} times'.format(hestr, 7)    # show integer with sign
     ==>    'Say Hello,   +7 times'
B186 >>>    'Say {}, {:-04d} times'.format(hestr, 7)    # show integer only with negative sign
     ==>    'Say Hello, 0007 times'
B187 >>>    'Say {}, {:-02d} times'.format(hestr, integ)    # again, no truncation
     ==>    'Say Hello, -234 times'
B188 >>>    '100 / 6 == {}'.format(100/6)          # default formatting for float
     ==>    '100 / 6 == 16.666666666666668'
B189 >>>    '100 / 6 == {:11.5f}'.format(100/6)    # first number is the total length
     ==>    '100 / 6 ==    16.66667'
B190 >>>    '100 / 8 == {:07.3f}'.format(100/8)    # second number is the decimal digits
     ==>    '100 / 8 == 012.500'
```