The information in this and the other summaries is largely a simplification of the official python3 documentation. For more detailed information, see `https://docs.python.org/3/`.

Some code examples contain features that have not yet been addressed in the respective week. They have been added for reference during later stages of the course and are marked with a gray background instead of light green.

Week 1

Printing to console

```
print()
```

Print objects to console. The objects to be printed will be interpreted as strings if necessary. If multiple arguments are given (separated by commas), they will be printed separated by spaces. By default, each print command also prints a newline after the input. This can be changed using the `end` keyword argument.

- **Print a string:**

  ```
  >>> print('Hello')
  Hello
  ```

- **Print a number:**

  ```
  >>> print(123)
  123
  ```

- **Print the result of an expression:**

  ```
  >>> print(3 + 5)
  8
  >>> print('Cat' + 'Dog')
  CatDog
  >>> print('Hello' * 5)
  HelloHelloHelloHelloHello
  ```

- **Print multiple items (may be of different types):**

  ```
  >>> print('Cat', 'Dog', 12, 3.5)
  Cat Dog 12 3.5
  ```

- **Print without a newline at the end:**

```
>>> print('Hello', end=' ')
Hello
```

- **Print without a newline, but with an other character at the end:**

```
>>> print('Hello', end='*')
Hello*
```

Binary operators

= Operator

Assigns the result of an expression on the right side to a variable on the left side. If the variable already contained a value, it is overwritten.

- **Arithmetic:**

```
x = 3 + 5.0
```

- **Strings:**

```
s = 'Cat'
```

- **Arithmetic:**

```
x = 3 + 5.0
```

- **Lists:**

```
x = [1, 2, 3]
```

+ Operator: `A + B`

If the expressions on either side are numbers, they are added algebraically. If they are strings or lists, they are concatenated. Both expressions should be of the same type.

- **Arithmetic addition:**

```
x = 3 + 5.0
```

- **String concatenation:** Note that this is the same as when used with `print()`, since `print()` interprets objects as strings.

```
x = 'Cat' + 'Dog'
```

- **List concatenation:**

```
x = [1, 2] + [3, 4]
```

`-` Operator: `A - B`

- **Arithmetic subtracation:**

```
x = 3 - 5.0
```

`*` Operator: `A * B`

Multiplication or repetition

- **Arithmetic multiplication:**

```
x = 3 * 5.0
```

- **Repeat a string 5 times:**

```
x = 'Five' * 5
```

`/` Operator: `A / B`

- **Arithmetic division:**

```
x = 3 / 5.0
```

`//` Floor division operator: `A // B`

Returns the integer part of the quotient. The resulting value is a whole integer, though the result's type is not necessarily int. Use it only to divide by integers; dividing by floats may give substantial errors.

- **Floor division:**

```
7 // 3
```

`%` Operator: `A % B`

Returns the remainder of the division `A / B`.

- **Modulo/Remainder:**

```
7 % 3
```

`**` Exponent operator: `A ** B`

Returns `A` raised to the power of `B`.

- **Raise 5 to the 3rd power:**

  ```
  5**3
  ```

- **9 to the 0.5th power (take square root):**

  ```
  2**0.5
  ```

Extended assignment operators: `+=, -=, *=, /=, //=, %=, **=`

These operators combine the binary operator they correspond to with an assignment to the first argument variable. This is often used to increment a counter in a loop.

- **Equivalent to `x = x + 1`**

  ```
  x += 1
  ```

- **Equivalent to `x = x // 3`**

  ```
  x //= 3
  ```

Comparison operators:

- **Equality**: ==
- **Not equal**: !=
- **Greater than**: >
- **Greater or equal**: >=
- **Less than**: <
- **Less or equal**: <=

Comparisons return boolean values: `True` or `False`. These operators compare the values of two objects. If both are numbers, they are converted to a common type. Comparison of objects depends on the type:

- Numbers are compared arithmetically.
- Strings are compared lexicographically using the numeric codes of their characters.
- Tuples and lists are compared lexicographically using comparison of corresponding elements. This means that to compare equal, each element must compare equal and the two sequences must be of the same type and have the same length.
- Dictionaries compare equal if and only if their sorted (key, value) lists compare equal.

These examples use `print()` to show the comparison result:

- **Compare numbers:**

  ```
  >>> print(1 == 2)
  False
  ```

- **Compare numbers and expressions:**

  ```
  >>> print(1 == 3 - 2)
  True
  ```

- **Compare integers and floats - arithmetically by value:**

  ```
  >>> print(1 == 1.0)
  True
  ```

- **Compare strings - capitalization matters:**

  ```
  >>> print('Cat' == 'Dog')
  False
  >>> print('abc' == 'Abc')
  False
  ```

- **Compare lists:**

  ```
  >>> print([1, 2, 3] == [1, 3, 2])
  False
  ```

- **Compare dictionaries:**

  ```
  >>> print({1:'Cat', 2:'Dog', 3:'Rabbit'} == \
  ... {2:'Dog', 1:'Cat', 3:'Rabbit'})
  True
  ```

Boolean operators:

AND operator

Logical AND (`and`) returns `True` if all the expressions in a statement are individually true:

```
>>> True and True
True
>>> True and False
False
>>> True and False and True
False
```

OR operator

Logical OR (`or`) returns `True` if any one of the expressions in a statement is individually true:

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> True or False or True
True
```

NOT operator

Negation (`not`) inverts the boolean value of the expression following it: `not False` is `True`, `not True` is `False`:

```
>>> True and not True
False
>>> True and not False
True
>>> False and not False
False
>>> True and not False and True
True
```

Data types

```
type()
```

Returns the data type of an object.

- **String:**

  ```
  >>> type('abc')
  <class 'str'>
  ```

- **Integers:**

  ```
  >>> type(1 + 2)
  <class 'int'>
  ```

- **Floats:**

  ```
  >>> type(2 + 0.5)
  <class 'float'>
  ```

- **Lists:**

  ```
  >>> type([1, 2, 3])
  <class 'list'>
  ```

- **Dictionaries:**

  ```
  >>> type({'key': 'value'})
  <class 'dict'>
  ```

```
float()
```

Returns a floating point number constructed from a number or string. If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be '+' or '-'; a '+' sign has no effect on the value produced.

- **From integer:**

  ```
  >>> float(123)
  123.0
  ```

- **from integer expression:**

  ```
  >>> float(3 + 5)
  8.0
  ```

- **From string, with and without '-' sign:**

  ```
  >>> float('-23')
  -23.0
  >>> float('+1.23')
  1.23
  ```

- **From string, including whitespace and newline character:**

  ```
  >>> float('   -12345\n')
  -12345.0
  ```

- **From string, in scientific notation:**

  ```
  >>> float('1e-003')
  0.001
  >>> float('+1E6')
  1000000.0
  ```

```
int()
```

Returns an integer constructed from a number or string. If **x** is a positive number, it will round it down, if **x** is negative, it will round it up. If **x** is not a number, then **x** must be a string representing a whole number.

- **Integers:**

  ```
  >>> int(123)
  123
  ```

- **Floats:**

  ```
  >>> int(5.5)
  5
  ```

- **Strings:**

  ```
  >>> int('40.44')
  ValueError: invalid literal for int() with base 10: '40.44'
  >>> int('23')
  23
  ```

```
str()
```

Returns a string representation of an argument.

- **Integers:**

  ```
  >>> str(10)
  '10'
  ```

- **Numeric expressions:**

  ```
  >>> str(22/3)
  '7.333333333333333'
  ```

- **Strings:**

  ```
  >>> str('43')
  '43'
  ```

- **Lists:**

  ```
  >>> str(['1', '2'])
  "['1', '2']"
  >>> str(['a', 'b'])
  "['a', 'b']"
  ```

- **Empty dictionary:**

  ```
  >>> str({})
  '{}'
  ```

User input

```
input():
```

Waits for a line of user input from the terminal. An optional message can be displayed first to prompt the user for input. Input is sent when the Enter key is pressed. The input is converted to a string (without a newline character at the end) and returned.

```
x = input('Enter password: ')
```

Import libraries (modules, packages)

```
import:
```

The basic import statement is executed in two steps:

1. Find a module, load and initialize it if necessary
2. Define a name or names that can be used in the current file to refer to it.

If the module name is followed by `as`, then the name following as must be used in the file as reference to the module. The `from` form (`import identifier from module`) uses a slightly more complex process and also checks whether the defined import can indeed be carried out. The identifier is the name that must be used as a reference. If the list of identifiers is replaced by a star ('*'), all public names defined in the module can be used in the file, but this is not recommended.

- **numpy imported, numpy is reference:**

  ```
  import numpy
  ```

- **numpy.random.randint imported, ri is reference:**

  ```
  import numpy.random.randint as ri
  ```

- **numpy.random.randint, randint is reference:**

  ```
  from numpy.random import randint
  ```

Flow Control

```
if-elif-else:
```

Provides conditional branching. Code blocks in an if statement will only execute if the condition evaluates to `True`. Code blocks in an `else` statement will only execute if the corresponding `if` condition evaluates to `False`. if-elif-else statements allow for multiple conditions, only one of which will execute.

- **Simple if statement:**

```
if x > 1000:
    print('x is big')
```

- **If-else statement, only one of two branchs will execute:**

```
if x >= 0:
    print('x is positive')
else:
    print('x is negative')
```

- **If-elif-else statement, only one of several branches will execute:**

```
if has_feathers(x) is True:
    print('x is a bird')
elif can_move(x) is True:
    print('x is an animal')
elif is_alive(x) is True:
    print('x is a plant')
else:
    print('x is probably a rock')
```

- **Check whether a list contains an object:**

```
if 'abc' in mylist:
    print('Element exists!')
else:
    print('Element not found...')
```

Week 2

Flow Control

for-Loop

Repeats the statements in a code block for each element in a sequence or container. The number of repetitions is known at the start of the loop.

- **Repeat for each whole number:**

```
for x in range(1, 11):
    print('Counted to: ', x)
```

- **Specify a different increment:**

```
for x in range(3, 10, 2):
    print('Uneven: ', x)
```

- **Repeat for a reversed range:**

```
for x in range(10, 0, -1):
    print('Countdown: ', x)
```

- **Process each element in a list:**

```
for x in ['cat', 'dog', 'bird']:
    print('Look at the ', x)
```

- **Process each element in a list using indexing:**

```
l = [0] * 4
for i in range(4):
    l[i] = (i + 1) ** 2
print(l)
```

while-Loop

Repeats the statements in a code block for as long as a condition is True. In this loop, the number of repetitions is not known in advance.
NOTE: Take care with the loop condition. It is possible to write while loops which never end ('infinite loops') and make a program appear to hang. It is also up to you to make sure that counters are correctly updated within the loop

- **Simple counting loop:**

```
x = 0
while x < 10:
    print('counted to: ', x)
    x += 1 # If this line were missing, it would cause an
    # infinte loop since x would never change!
```

`break` and `continue`

These statements allow loops to be terminated from within, or for certain loop iterations to be skipped.

- **Stopping a loop early if a condition becomes `True`, used to find the first number >= 10000 which is divisible by 1234:**

```
x = 10000
while True: # loop will run forever unless something stops it
    if x % 1234 == 0: # check for zero division remainder
        print('Found solution: ', x)
        break # this stops the while loop
    x+=1
```

- **Skipping a loop iteration and continuing with the remaining iterations:**

```
for x in range(-3, 3):
    if x == 0: # skip case which would cause division-by-zero
        continue
    print('1/' + str(x) + ': ', 1.0 / x, end='; ')
```

combining `for` and `if`:

- **An `if` statement can be placed within a loop:**

```
for i in range(7):
    if i % 2 == 0:
        print(i, end=' ')
```

Week 3

Flow Control

Nested Loops

A loop is placed inside another loop (and another loop can be placed inside this nested loop again etc.)

- **Repeat for each whole number:**

```
for x in ['a', 'b', 'c']:
    for y in range(1, 4):
        print (x, y, end='; ')
```

Variable Loops

Instead of hard-coding the number of repetitions in a for-loop using a constant, the number of repetitions can be made flexible by using a variable.

- **Loop repetitions based on user input:**

```
n = int(input('Which factorial?'))
fact = 1
for i in range(1, n+1):
    fact = fact * i
print(fact)
```

Random Numbers

```
numpy.random.randint()
```

Returns randomly chosen whole numbers from a given interval. The lower bound is included and may be returned. The upper bound is excluded and will never be returned. Values are drawn from a uniform distribution.

- **Return a single random number from the list [1, 2, 3]:**

```
numpy.random.randint(1, 4)
```

```
numpy.random.seed()
```

Seeds the generator, so that a program can use the same sequence of random numbers each time it is run. Takes an integer as argument.

Use of backslash

Escape Characters

Certain characters which are commonly used do not have a character symbol, or that symbol may already have a meaning in python code. Examples are a tab, backslash or newline/line-break. These characters have reserved "escape sequences" which allow them to be used like normal characters. The backslash character '\' is used in python to identify the start of an escape sequence. This means that writing an actual backslash requires its own escape sequence (commonly used when writing directory paths as strings).

- **Printing tab characters using the escape sequence '`\t`':**

```
print('Some words\tseparated by\ttabs')
```

- **Printing newline characters using the escape sequence '`\n`':**

```
print('Some\nmore\nwords')
```

Line Continuation

Some python statements may become too long to display on a screen or on paper. Line continuation allows long statements to be split onto multiple lines by placing a single backslash character '\' at the end of the line and continuing the statement on the following line.

- **Without line continuation:**

```
if has_fur == True and number_of_ears == 2 and weight_kgs < 10 and numb
    print("I'm a cat")
```

- **With line continuation:**

```
if has_fur == True \
and number_of_ears == 2 \
and weight_kgs < 10 \
and number_of_legs == 4 \
and teeth == 'sharp' \
and likes_mice == True:
    print("I'm a cat")
```

Week 4

No new concepts: Combining if-statements, loops, variables of a few different types and operators form the core of programming and allows for countless possibilities for different programs. However, this requires practice.

Week 5

Lists

Lists are mutable ordered sequences of values. Lists may be any length, may be extended or shortened dynamically and the same list can contain different types of values. A list may be empty. Duplicate values are allowed. Lists may contain other lists.

- **Create an empty list:**

  ```
  mylist = []
  ```

- **Create a list with initial values:**

  ```
  mylist = [1, 2, 3, 'bear']
  ```

- **Create a list of a certain length n:**

  ```
  mylist = [0]*n
  ```

- **Create a list containing the first 5 whole numbers (starting from 0!):**

  ```
  mylist = list(range(5))
  ```

- **Create a list of lists representing a 3x3-matrix. The inner lists represent the rows:**

  ```
  mymatrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
  ```

Indexing

Lists are accessed through an index which starts at 0. For example, given the list [10, 20, 30, 40, 50], the value at index 2 is 30. Elements may be accessed from the back of the list by using negative index values. If a list contains N items, a value used to index the list may not be larger than N-1, and not smaller than -N. Elements in lists-of-lists are accessed by specifying each index individually.

- **Access the 3rd element in a list (note zero-based indexing!):**

  ```
  mylist[2]
  ```

- **Assign a new value to the 3rd element:**

  ```
  mylist[2] = 3.1415
  ```

- **Access the last element in the list:**

  ```
  mylist[-1]
  ```

- **Create a list of lists representing a 3x3-matrix. The inner lists represent the rows:**

```
mymatrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- **Error (index too large):**

```
[1, 2, 3][3]
```

- **Error (index too small):**

```
[1, 2, 3][-4]
```

- **Access the element in the top right corner (row 1, column 3) of a 3x3-matrix:**

```
mymatrix[0][2]
```

Slicing

Slicing syntax allows ranges of elements in a list (and other sequences, such as strings and tuples) to be accessed at once. The end value specified is the index of the first element NOT to be included in the result. Slices may be taken from the end of the list by using negative indices. The following examples assume that `start` and `end` are valid indices.

- **Return a copy of the whole list:**

```
mylist[:] #Note that this creates an independent copy,
          #in contrast to: yourList = myList
```

- **Return a list containing items start through end-1:**

```
mylist[start:end]
```

- **Return a list containing items start through the end of the list:**

```
mylist[start:]
```

- **Return a list containing items from the beginning of the list through end-1:**

```
mylist[:end]
```

- **Return a list containing the last two items:**

```
mylist[-2:]
```

- **Return a list containing everything except the last two items:**

```
mylist[:-2]
```

- **Return a list containing everything except the first and last items:**

```
mylist[1:-1]
```

The following examples do not assume that start and end value are valid indices: they are positive integers but may not be between 0 and the length of the list minus 1:

- **Return a list containing the last two items:**

  ```
  yList[len(myList)-3:len(myList)+2]
  ```

- **Return an empty list if start is larger than length of the list minus 1:**

  ```
  myList[len(myList):len(myList)+2]
  ```

Membership testing

The membership operator 'in' evaluates to true if it finds a variable in the specified sequence (such as a list) and false otherwise.

```
if 's' in ['t', 's', 'v']:
    print ('present')
```

List functions

```
len()
```

Returns the number of items in a list (or other objects, such as strings, tuples and dictionaries).

- **Get the number of elements in a list:**

  ```
  len([4, 5, 6, 7, 8])
  ```

- **Get the number of key-value pairs in a dictionary:**

  ```
  len({1:'a', 2:'b', 3:'c'})
  ```

```
append()
```

Extends a list by adding a new element to the end of the list.

- **Append a string to an existing list:**

  ```
  mylist.append('Cat')
  ```

```
extend()
```

Extends a list by adding all elements from an existing list to the end of the list, in order.

- **Append three strings from a list to the end of an existing list:**

  ```
  mylist.extend(['a', 'b', 'c'])
  ```

```
insert()
```

Inserts a new element into a list at a given index position, in front of the current value at that position.

- **Insert 2018 at index 3. the current value at index 3 will appear after 2018:**

  ```
  mylist.insert(3, 2018)
  ```

```
remove()
```

Removes the first occurrence of a given object from a list. Subsequent occurrences will not be affected. If no occurrence is found, an error is raised.

- **Remove a string from a list:**

  ```
  mylist.remove('bear')
  ```

```
del
```

Removes the element at a given index from a list. If the index is not valid, an error is raised.

- **Remove the 4th element:**

  ```
  del mylist[3]
  ```

```
pop()
```

Removes and returns the item at a given index in a list. If no index is given, the last element is removed and returned. If the index is not valid, an error is raised.

- **Remove and return the last element:**

  ```
  mylist.pop()
  ```

- **Remove and return the second last element:**

  ```
  mylist.pop(-2)
  ```

```
index()
```

Finds the index of the first occurrence of a given value in a list. If the value is not found, an error is raised.

- **Find the index of an item:**

  ```
  ['a', 'e', 'i', 'o', 'u'].index('o')
  ```

- **Error, item not found:**

  ```
  ['a', 'e', 'i', 'o', 'u'].index('x')
  ```

- **Only the first index is returned if there are multiple copies:**

  ```
  [1, 2, 3, 1].index(1)
  ```

```
reverse()
```

Reverses the order of elements in the list.

- **Reverse an existing list:**

  ```
  mylist.reverse()
  ```

```
sort()
```

Sorts the elements of a list in ascending order. The list is sorted in place. Strings are sorted lexicographically in ascending order (try it with `mylist = ['abc', 'bc', '123', 'ABC', 'aBC', 'a', 'b']`) All list elements must be comparable. The sort order may be reversed by setting the `reversed` keyword to `True`.

- **Sort in ascending order:**

  ```
  mylist.sort()
  ```

- **Sort in descending order:**

  ```
  mylist.sort(reversed=True)
  ```

```
sorted()
```

Returns a COPY of the original list with the elements sorted. Use this instead of .sort() if you need access to both the sorted and unsorted lists.

- **Return a sorted copy of an existing list:**

  ```
  sorted_list = sorted([3, 4, 1, 6])
  ```

```
enumerate()
```

Returns a tuple containing a count (from 0) and a value obtained from iterating over a list (or another iterable like a string or a tuple).

- **Print index and value of elements in a list:**

  ```
  for i,value in enumerate(myList):
      print(i, value)
  ```

  is equivalent to:

  ```
  for i in range(len(myList)):
      print(i, myList[i])
  ```

```
zip()
```

Aggregates elements from multiple lists. For example, if the lists col1 ... colN each contain the data from one column in a table, then `list(zip(col1, col2, ... colN))` will return a list of rows from the table. The elements of the returned list are tuples, with elements ordered as the columns were.

- **Get a list of tuples with the letter value first:**

  ```
  list(zip(['a', 'b', 'c'], [1, 2, 3])
  ```

- **Get a list of tuples with the number value first:**

  ```
  list(zip([1, 2, 3], ['a', 'b', 'c'])
  ```

Copies and Synonyms

```
copy.deepcopy()
```

By default, assignment of complex objects (such as a class or nested list) does not make a copy of the original object but just assigns a reference to it. This means that multiple variables can point to the same object, and using any of them to change the object will make all of them change. If an identical but distinct clone of an object is needed, it must be deep copied (copying not just the reference to the object but the object itself). Simple types (numbers, strings, booleans, etc.) are always cloned at assignment, they do not need to be deep copied.

- **Assignment of complex objects only copies a reference to the original object:**

  ```
  a = [[1,2,3], 4]
  b = a
  b[0] = [5,6,7]
  print(a)
  ```

- **Deep copying complex objects produces an identical clone:**

  ```
  import copy
  a = [[1,2,3], 4]
  b = copy.deepcopy(a)
  b[0] = [5,6,7]
  print(a)
  ```

- **Alternative for lists (but not lists of lists):**

  ```
  lnew = l[:]
  ```

- **Alternative for numpy arrays:**

  ```
  anew = 1*a
  ```

Tuples

Like lists, tuples are an ordered sequence of values. Unlike lists, tuples are immutable und cannot be modified once they are created. This allows them to be faster and more efficient than lists if modification is not needed. An example of where to use tuples would be for storing data from one row in a table where the data values are unlikely to change while the program is running, for example: (firstname, lastname, birthdate, major). Data for the whole table would be stored as a list of such tuples, so that it can be easily sorted or have whole tuples added/removed.

- **Create a tuple:**

```
mytuple = ('John', 'Smith', '2000/01/01', 'Biology')
```

- **Creating a tuple with a single value still requires a comma:**

```
('abc',)
```

- **Create a tuple implicitly:**

```
mytuple = 1, 2, 3, 4, 5
```

- **Accessing tuple elements is similar to lists:**

```
mytuple[1]
```

- **Tuple values can be unpacked into separate variables:**

```
fname, lname, date, major = mytuple
```

- **Tuples can be sliced like lists:**

```
fname, lname = mytuple[0:2]
```

- **Tuple syntax allows multiple assignment:**

```
x, y = 10, 20
```

- **Tuple syntax allows swapping values in place:**

```
x, y = y, x
```

File input/output

```
open()
```

Opens a file on disk and returns a file object which can be used to access the file. Will raise an error if the file could not be opened. The allowed actions for the file (reading, writing, etc.) are specified by a mode argument. Opening the file does not read or write data. Commonly used modes are:

- `'r'`: Open for reading (default)
- `'w'`: Open for writing. Existing contents will be overwritten
- `'a'`: Open for writing, appending to the end of the file if it exists

NOTE: A file opened using `open()` must be manually closed after use by calling `close()` on it to ensure that all data is written correctly and access to the file is not blocked.

- **Implicitly open a file for reading only:**

  ```
  myfile = open('filename.txt')
  ```

- **Explicitly open a file for reading only:**

  ```
  myfile = open('filename.txt', 'r')
  ```

- **Open a file for writing only:**

  ```
  myfile = open('filename.txt', 'w')
  ```

```
.read()
```

Reads data from an open file and returns them as a string. If called again after the end of the file is reached, an empty string is returned.

- **Read the entire contents of an open file into a string:**

  ```
  data = file.read()
  ```

```
.readline()
```

Reads the next line of text from an open file and returns it as a string. A line number can be specified to return only that line.

- **Read the first or next line from an open file:**

  ```
  myfile.readline()
  ```

- **Read the third line from an open file:**

  ```
  data = myfile.readline(3)
  ```

```
.readlines()
```

Reads all lines from an open file and returns them as a list of strings.

- **Read the entire contents of an open file:**

  ```
  list_of_lines = myfile.readlines()
  ```

```
.write()
```

Writes string data to an open file. Other data types must be converted to strings before writing them. Whether the file is overwritten or appended to will depend on the mode it was opened with.

- **Write a string to an open file:**

  ```
  myfile.write('Some important data to save')
  ```

```
.close()
```

Closes an open file object and releases the file on disk for other programs to use. Any data which was not yet written to disk will be written before closing.

- **Close a previously opened file:**

  ```
  myfile.close()
  ```

String Processing

`split()`

Returns a list of 'words' in the string. Each 'word' is the sequence of characters between occurrences of a delimiter string. For natural text, the delimiter would be `' '`. If `split()` is called without argument, the words are separated by arbitrary strings of whitespace characters (eg. space, tab, and newline).

- **Splitting natural text:**

  ```
  'The words we write'.split(' ')
  ```

- **Splitting CSV data:**

  ```
  '1/4/2014,9,1,blue'.split(',')
  ```

- **Splitting spaced CSV data:**

  ```
  '1/4/2014, 9, 1, blue'.split(', ')
  ```

- **Splitting an empty string:**

  ```
  ''.split('.')
  ```

- **Splitting without using an argument:**

  ```
  'Remove   whitespace \tcharacters\n'.split()
  ```

`join()`

Returns a string which is the concatenation of the strings in a list, with a given separator string. The list must contain only string types. The delimiter is only added between strings from the list.

- **Concatenating with no separator:**

  ```
  ''.join(['a', 'b', 'c'])
  ```

- **Concatenating with commas:**

  ```
  ', '.join(['a', 'b', 'c'])
  ```

- **Arbitrary separator:**

  ```
  ', followed by: '.join(['a', 'b', 'c'])
  ```

```
strip()
```

Removes leading and trailing characters from a string. All combinations of the characters are removed. Characters within the string (i.e. 'protected' on both sides by characters which will not be removed) will not be affected. If no argument is given, any whitespace is removed, including spaces, tabs, and newlines (`\n` for mac; `\r\n` for windows).

- **Remove whitespace characters:**

  ```
  '   spacious   \n'.strip()
  ```

- **Character order is ignored:**

  ```
  'www.example.com'.strip('cmowz.')
  ```

- **Characters within the string are ignored, even if they would otherwise be removed:**

  ```
  '#....... Section 3.2.1 Issue #32 .......'.strip('.#! ')
  ```

Week 6

String Formatting

```
format()
```

Values can be inserted into strings using format(), however you may also encounter the less-capable old syle which uses `%`. See https://pyformat.info/ for examples of both styles. The new style uses `''` placeholders in strings to indicate where values are to be inserted. These placeholders can be customized to show the values in a certain format.

- **Basic use:**

  ```
  '1 {} 3 {}'.format(2, 4)
  ```

- **Formatting integer numbers as integer:**

  ```
  '{:d}'.format(123)
  ```

- **Formatting floating point numbers:**

  ```
  '{:f}'.format(123.5)
  ```

- **Formatting integers as floating point numbers:**

  ```
  '{:f}'.format(12)
  ```

- **Formatting floating point numbers using e-notation:**

  ```
  '{:e}'.format(123.5)
  ```

Numbers can be padded/aligned within a specific width. This works the same for all types of numbers.

- **Right alignment in a 10-character string:**

  ```
  '{:10d}'.format(12)
  ```

- **Left alignment in a 10-character string:**

  ```
  '{:<10d}'.format(12)
  ```

- **Left alignment, padded with a non-whitespace character:**

  ```
  '{:x<10d}'.format(12)
  ```

- **Formatting integers as floating point numbers:**

  ```
  '{:f}'.format(12)
  ```

- **Truncating for float and e-notation:**

```
'{:.2f}'.format(3.14159)
```

- **Padding with leading zeros combined with truncating to 2 digits after the decimal point:**

  ```
  '{:06.2f}'.format(3.14159)
  ```

`upper(), lower()`

Replaces all lowercase (uppercase) letter characters in a string with their uppercase (lowercase) equivalents. Other characters are not changed.

- **Convert lowercase letters to uppercase:**

  ```
  '123_abc.+-=_ABC'.upper()
  ```

- **Convert uppercase letters to lowercase:**

  ```
  '123_abc.+-=_ABC'.lower()
  ```

`partition()`

Splits the string at the first occurrence of a separator string, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

- **Partition on the first occurrence of a character:**

  ```
  'www.example.com'.partition('.')
  ```

- **Partition on the first occurrence of a string:**

  ```
  'www.example.com'.partition('example')
  ```

```
replace()
```

Returns a copy of the string with all occurrences of a given substring replaced by a new substring.

- **Replace '-' characters:**

  ```
  '2018-01-05'.replace('-', ' ')
  ```

- **Extend existing whitespace:**

  ```
  'way too much space !'.replace(' ', '    ')
  ```

- **Substitute a word:**

  ```
  'Cats or dogs'.replace('or', 'and')
  ```

Dictionaries

Dictionaries are an unordered collection of key→value pairs. Because it is unordered, iterating over the dictionary may return data in a different/random order. In future python versions, the elements in a dictionary will be ordered according to the order they were inserted but don't count on this for now. Any type of object can be used as a key in a dictionary, as long as it is immutable (like numbers, strings or simple tuples). Values can be any type of object, including for example strings, floats, lists, lists of lists, dictionaries, etc. Both keys and values can be of different types within the same dictionary. Dictionaries are useful when values will be accessed by a name or ID key instead of a sequence index number. Keys must be unique within a dictionary, but duplicate values are allowed.

- **Create an empty dictionary (brace notation):**

  ```
  mydict = {}
  ```

- **Inserting a new key-value pair (if the key already exists, its value is overwritten instead):**

  ```
  mydict['abc'] = 'alphabet'
  ```

- **Get all the keys in a dictionary (order can vary):**

  ```
  mydict.keys()
  ```

- **Get all values in a dictionary (order can vary):**

  ```
  mydict.values()
  ```

- **Create a dictionary with initial data and <key>:<value> syntax:**

  ```
  word_lengths = {'hello': 5, 'cat': 3, 'orange': 6}
  ```

- Return the data stored for key `'cat'` (This will cause an error if the key is not found):

```
x = word_lengths['cat']
```

- Delete a key and it's value from a dictionary (This will cause an error if the key is not found):

```
del word_lengths['cat']
```

- Check whether a key is present using the **`in`** keyword:

```
if 'hello' in word_lengths:
    print("'hello' is a key in word_lengths")
```

- Loop through keys using the **`in`** keyword:

```
for word in word_lengths:
    print(word, end=' ')
```

Week 7

Functions

Functions are named code blocks which represent a coherent unit of a program. They can be given input data and can return output data. A common use for functions is to provide frequently used operations in a simple form and avoid duplicating code. Many built-in operations in python are implemented as functions, for example print() or len(). Functions are defined with the def keyword, a unique name and a list of input arguments. If a function contains a return statement, the function will terminate when it reaches it and the value(s) in this statement will be made available outside the function. Function arguments can be assigned by the position in which they appear in the function definition.

- **Simple function to calculate the cube value of a number and return it:**

```
def calculate_cube(x):
    cube = x * x * x
    return cube

cb = calculate_cube(8)
print(cb)
```

- **A function that takes more than one argument and returns more than one variable by using a single tuple:**

```
def product_and_sum(x,y):
    sum = x + y
    product = x * y
    return sum, product

print(product_and_sum(2, 3))
s, p = product_and_sum(2, 3)
print(s)
print(p)
```

Scope of variables and the use of global:

In Python, variables follow these rules:

- When we create a variable inside a function, it's local by default (it will not be defined or accessible outside that function).
- When we create a variable outside of a function, it's global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function, instead of creating a new local variable with the same name.
- Using the global keyword outside a function has no effect.

WARNING: Over-using globals increases the likelihood of unintended consequences and makes code harder to understand and debug. Before using the global keyword in a function, consider whether the same result can be achieved using only function arguments, local variables and return statements!

- Creating a global variable and accessing it locally (in a function):

```
g = 1 # global variable

def local_print():
    print('local: ', g) # local access

local_print()
print('global: ', g) # g should still be 1
```

- Global variables cannot be modified locally by default:

```
g = 1 # global variable

def add():
    # increment g locally. This will cause an error because
    # there is no local variable called g:
    g += 1
    print('local: ', g) # local access

add()
print('global: ', g) # g should still be 1
```

- Using the global keyword to modify a global variable locally. Changes remain after the function finishes:

```
g = 1 # global variable

def add():
    global g # get access to global variable
    # increment g locally. This will cause an error because
    # there is no local variable called g:
    g += 1
    print('local: ', g) # local access

add()
# g will have changed. If we want to know why or how, we have
# to look at all functions which declare g global.
print('global: ', g)
```

- Using function arguments, local variables and return statements to modify a global variable instead of using the global keyword. This makes the code easier to understand:

```
g = 1 # global variable

def add(number): # the function requires an input value
    # increment the number locally. This will not affect
    # anything outside the function:
    number += 1
    print('local: ', number) # local access
    # return a modified value which can be used to overwrite a
```

```python
    # global variable, but doesn't automatically overwrite it
    # like before:
    return number

# pass the value of g to the function and explicitely use the
# returned value to overwrite g.
g = add(g)
# g will have changed, but now it's clearer why and how that
# happened:
print('global g: ', g)
```

Week 8

NumPy is a package for fast and efficient numerical calculations. It is less flexible but much faster (10-1000x, depending on the problem) than using pure python constructs when working with large amounts of data.

Arrays

NumPy arrays are similar to simple or multidimensional python lists. They are optimized for speed in numeric computation. Unlike lists, arrays can only store a single type of data, and have a fixed size. Since arrays can only store a single type, they handle calculations with integers and float not exactly the same as lists and more attention must be payed to the data type. Array elements are accessed by indices or slicing like with lists, but provide more possibilities for slicing in multiple dimensions. Indices are zero-based like with python lists. Arithmetic operations between arrays are possible if the dimensions of the arrays match, and are performed element-wise.

- **Create a 1D array from a list:**

  ```
  numpy.array([1, 2, 3, 4])
  ```

- **Create a 2D array from a nested list:**

  ```
  numpy.array([[1, 2], [3, 4]])
  ```

- **Create a 2D array from a nested list with a specified data type:**

  ```
  numpy.array([[1, 2, 3], [4, 5, 6]], np.int32)
  ```

- **Access the element in the second row and fourth column of a 2D array:**

  ```
  myarray[1, 3]
  ```

- **Slicing a 1D array to get a subarray:**

  ```
  numpy.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])[1:5]
  ```

- **Slicing a 2D array to get a (2D) subarray:**

  ```
  numpy.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])[1:3, 0:2]
  ```

- **Arithmetic operations on arrays (same for `-` and `*` and `/`):**

  ```
  numpy.array([1, 2, 3, 4]) + np.array([2,2,2,2]))
  ```

- **The data type does not automatically change, here `a` will still be an array of integers:**

  ```
  a = numpy.array([1, 2, 3])
  a[0] += 0.2
  ```

NumPy Functions

```
numpy.linspace()
```

Return a 1D array of evenly spaced numbers over a specified interval. By default, the first and last values returned are the specified bounds of the interval.

- **Return an array of 5 numbers from 2 to 3:**

  ```
  np.linspace(2, 3, 5)
  ```

- **Return an array of 100 numbers from 5 to 50:**

  ```
  np.linspace(5, 50, num=100)
  ```

```
numpy.array()
```

Returns an array or multidimensional array from an existing data sequence. Input values need not have the same type, but all values will be converted to the most general type.

- **Create a 1D array of integers from a list:**

  ```
  numpy.array([1, 2, 3])
  ```

- **Create a 1D array of floats from a list of mixed types:**

  ```
  numpy.array([1, 2, 3.0])
  ```

- **Create a 2D array from a nested list:**

  ```
  numpy.array([[1, 2], [3, 4]])
  ```

- **Create a 1D array with a specified data type:**

  ```
  numpy.array([1, 2, 3], dtype=float)
  ```

```
numpy.zeros()
```

Returns an array with a specified shape, filled with zeros.

- **Create a 1D array of 5 zeros:**

  ```
  numpy.zeros(5)
  ```

- **Create a 1D array of 5 zeros with a specified data type:**

  ```
  numpy.zeros((5,), dtype=int)
  ```

- **Create a 2D array of zeros (Note additional tuple brackets):**

  ```
  numpy.zeros((2, 3))
  ```

- **Create a 2D array of zeros with a specified data type:**

  ```
  numpy.zeros(shape=(3, 4), dtype=float)
  ```

```
numpy.ndarray.shape
```

Returns a tuple containing the dimensions of the array. Note this is a property, not a function and it therefore does not need brackets.

- **Get the shape of a 1D array with 4 elements:**

  ```
  numpy.array([1, 2, 3, 4]).shape
  ```

- **Get the shape of a 2D array:**

  ```
  numpy.array([[1, 2, 3], [4, 5, 6]]).shape
  ```

- **Get the shape of an array assigned to a variable:**

  ```
  myarray = numpy.zeros((3, 4))
  myarray.shape
  ```

Matplotlib

Matplotlib is a package for plotting and displaying data in various formats. The information here is largely a simplification from information at `matplotlib.org`

Import Statements

Matplotlib must be imported before it can be used:

- **Import matplotlib to generate and display figures, here:**

  ```
  import matplotlib.pyplot as plt
  ```

- **Import tools to save figures as image files, here:**

  ```
  import matplotlib.image as img
  ```

```
plt.plot()
```

Plots one or more Y-axis arrays against X-axis arrays, as lines or using marker symbols. Corresponding X- and Y-arrays must have the same length. If only one array is given, it is plotted on the Y axis against its own indices. Line styles and colors can be specified. To plot multiple series in the same graph, just call `plot()` once for each series before showing a figure or changing the active subplot. See `https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html` for detailed styling options and examples.

- **Plot array `y` against array `x` using default line style and color:**

  ```
  plt.plot(x, y)
  ```

- **Plot array `y` against array `x` using blue circle markers:**

  ```
  plt.plot(x, y, 'bo')
  ```

- **Plot array `y` against its own indices using default styling:**

  ```
  plt.plot(y)
  ```

- **Plot array `y` against its own indices using circle markers:**

  ```
  plt.plot(y, 'o')
  ```

- **Plot `y` against `x` using explicit styling keywords:**

  ```
  plt.plot(x, y, color='green', marker='o', linestyle='dashed',
           label='MyLabel') #label: see plt.legend()
  ```

```
plt.errorbar()
```

Plots one or more Y-axis arrays against X-axis arrays, as lines or using marker symbols with attached error bars. Corresponding X- and Y-arrays and their associated error bar arrays must have the same length. See `plot()` for styling options.

- **Plot array `y` against array `x`, with y-errorbars specified in array `yerr`, using default styling:**

  ```
  plt.errorbar(x, y, yerr)
  ```

- **Plot `y` against `x` with error bars along both x and y (note order!):**

  ```
  plt.errorbar(x, y, yerr, xerr)
  ```

```
plt.xlim(), plt.ylim()
```

Gets or sets the X and Y interval limits to display for the current axes, respectively.

- **Get the current X axis limits:**

  ```
  xmin, xmax = plt.xlim()
  ```

- **Get the current Y axis limits:**

  ```
  ymin, ymax = plt.ylim()
  ```

- **Change the current X axis upper limit only:**

  ```
  plt.xlim(xmax=3)
  ```

- **Change the current Y axis lower limit only:**

  ```
  plt.ylim(ymin=1)
  ```

```
plt.xlabel(), plt.ylabel()
```

Sets the label to display on the X and Y axis of the current axes, respectively.

- **Set the X axis label:**

  ```
  plt.xlabel('My x-label')
  ```

- **Set the Y axis label:**

  ```
  plt.ylabel('My y-label')
  ```

```
plt.legend()
```

Adds a legend to the current axes. The legend labels can be automatically determined, or explicitly given. It is recommended to assign each data series a label when plotting it, and then let `legend()` detect them automatically.

- **Add a legend which uses automatically detects inline labels that were assigned to the data series:**

  ```
  plt.plot([1, 2, 3], label='Inline label')
  plt.legend()
  ```

- **Add a legend with explicitly defined labels:**

  ```
  plt.plot([1, 2, 3])
  plt.legend(['A simple line'])
  ```

```
plt.title()
```

Sets the title of the current axes.

- **Set the plot title:**

  ```
  plt.title('My Title')
  ```

```
plt.show()
```

Displays a figure and waits for the figure window to be closed before the program continues. This function should be called after a plot has been created and customized, and is ready to display.

- **Display the current plot:**

  ```
  plt.show()
  ```

```
plt.figure()
```

Creates a new figure, which is a container for plots, subplots, axes, etc. Call this for each separate display window to be shown. If you are creating many figures, make sure to explicitly call `close()` on the figures you are not using to allow memory to be properly released. Each figure created has an index value. Calling `figure()` with no argument will auto-increment the index. Calling `figure()` with an existing index will make that figure active. If a string is given instead of an index, this is used as the window title for the figure.

- **Create a new figure:**

  ```
  plt.figure()
  ```

- **Create a new figure or activate the existing figure with index 1:**

  ```
  plt.figure(1)
  ```

- **Create a new figure with a given window title:**

  ```
  plt.figure('My Title')
  ```

```
plt.gca()
```

Get the current Axes instance on the current figure.

- **Get the current axes on the current figure:**

  ```
  plt.gca()
  ```

- **Assign the current axes to a variable:**

  ```
  ax = plt.gca()
  ```

```
axis.set_aspect()
```

Set the aspect ratio of the axis scaling (the ratio of y-unit to x-unit).

- **Automatic scaling, fills the rectangle with data:**

  ```
  #ax: variable assigned as under plt.gca()
  ax.set_aspect(aspect='auto')

  plt.gca().set_aspect(aspect='auto') # alternative
  ```

- **Equal scaling, `x` and `y` use same unit (same as `aspect=1.0`):**

  ```
  ax.set_aspect(aspect='equal')
  ```

- **Custom scaling factor, a circle will be stretched such that the height is `0.8` times the width:**

  ```
  ax.set_aspect(aspect=0.8)
  ```

```
axis.xaxis.set_visible()
```

Sets whether the axis will be visible or not. Parameter values are `True` or `False`. Passing `False` will hide the axis.

- **Hide X axis:**

  ```
  #ax: variable assigned as under plt.gca()
  ax.xaxis.set_visible(False)
  ```

- **Hide Y axis:**

  ```
  ax.yaxis.set_visible(False)
  ```

```
plt.imshow()
```

Displays a (2D) NumPy array as an image on the current axes.

- **Plot `myarray` as an image:**

  ```
  plt.imshow(myarray)
  ```

```
img.imsave()
```

Saves the data from a numpy array as an image file. The necessary output format is inferred from the filename extension but may be explicitly overwritten using format.

- **Save the plot of an array in PNG format:**

  ```
  plt.imsave('result.png', myarray)
  ```

- **Save the plot of an array using a specified format:**

  ```
  plt.imsave('result.jpg', myarray, format='JPG')
  ```

Week 9

Infinite and Not a Number

The NumPy library handles numbers slightly different than standard python. Standard python returns an error if a number is divided by zero or zero is divided by zero. In contrast, NumPy returns `inf` and `nan` in these cases, respectively. The NumPy library contains several functions that ignore `nan`. This can for example be practical when data points are missing.

```
numpy.nan
```

Returns a floating point representation of Not a Number. `NaN` and `NAN` are aliases of `nan`.

- **Creating an array containing `nan`:**

  ```
  myarr = numpy.array([1., 0., numpy.nan, 3.])
  ```

- **Creating an array containing only `nan`:**

  ```
  myarr = numpy.zeros((3,4)) + numpy.nan
  ```

- **Comparing two `nan` values always yields `False`:**

  ```
  np.nan == np.nan
  ```

Week 10

Speed: Dictionaries

Dictionaries are implemented in such a way that membership testing (testing whether a certain key is present) is very fast. Especially for larger dictionaries, this is much faster than testing whether a certain value is present in a dictionary or whether a certain element is present in a list.

Week 11

Speed: Array indexing

Using loops takes a relatively long time in python. Array indexing can prevent using loops and thus increase speed.

Integer array indexing

An array of indices (integers) can be passed to an array to access multiple array elements at once. Integer array indexing thus allows for selection of arbitrary items in an array.

- **Accessing elements of a NumPy (imported as `np`) array using another array as indices:**

```
a = np.array([8, 4, 3, 9, 2])
b = np.array([2, 0, 2, 1])
c = a[b] # c will be [3 8 3 4]
```

- **Summarize arrays with more than 10 elements by printing only the first and last 4 elements:**

```
np.set_printoptions(threshold=10, edgeitems=4)
```

Boolean array indexing

Boolean arrays can be used as masks to select particular elements in an array.

- **Creating a new array:**

```
a = np.array([6, 7, 5])
bo = np.array([True, False, True])
c = a[bo] # c will be [6, 5]
```

- **Creating a new array using a comparison (assume `a` and `b0` as above):**

```
d = a[a>5] # d will be [6, 7]
```

- **Changing specific elements inside an array using a comparison (assume `a` and `b0` as above):**

```
a[a<7] += 2 # a will be [8 7 7]
```

- **Changing specific elements inside an array using a comparison with another array (assume `a` and `b0` as above):**

```
d[c==5] *= 2 # d will be [6 14]
```

Basic mathematical functions

These functions have occurred several times during the course and can be used on lists and one-dimensional arrays for example.

```
sum()
```

Adds the values of a collection of numbers and returns the total. The values in the collection should all be numbers. If the collection contains both integers and floats, the result will be a float.

- **Return the sum total of all values in a list:**

  ```
  sum([1, 2, 3.0])
  ```

- **Return the sum total of all values in a tuple:**

  ```
  sum((1, 2, 3.0))
  ```

```
max(), min()
```

Return the largest (smallest) element in a collection or the larger (smaller) of two objects. Empty collections will raise an error unless a default value is given. If several values are maximal, the first one to be encountered is returned.

- **Return the larger of two numbers:**

  ```
  max(3.0, 5)
  ```

- **Return the smallest element in a list:**

  ```
  min([5, -3, 1.2, 12])
  ```

Week 13

```
numpy.bincount()
```

Counts the number of occurrences of each value in array of non-negative integers and returns an array containing the binned counts. The number of bins (and therefore the size of the returned array) is one larger than the largest value in the input. When two arrays of the same size are given as arguments, `bincount` can be used to calculate sums over variable-size chunks of the second input array using the indices in the first array.

- **Numpy (imported as `np`) array containing no duplicate values:**

  ```
  np.bincount(np.array([0, 1, 2, 3, 4]))
  ```

- **Numpy (imported as `np`) array containing several duplicate values:**

  ```
  np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
  ```

- **Numpy (imported as `np`) array constructed using array indexing with another array:**

  ```
  a = np.array([2, 4, 1, 4])
  b = np.array([0.1, 0.3, 1.2, 0.5])
  c = np.bincount(a, b)
  # eg. c[4] will be sum(b[a==4]), which is 0.3 + 0.5.
  ```

```
numpy.ndarray.astype()
```

Returns a copy of a given array, with all elements converted to a specified data type.

- **Get a copy of an array of floats, converted to integers:**

  ```
  numpy.array([1.3, 2.0, 2.5]).astype(int)
  ```

- **Get a copy of an array of integers, converted to floats:**

  ```
  numpy.array([3, 4, 5, 6]).astype(float)
  ```

Weeks 12 and 14

Combining known concepts and revision: no new programming concepts.