Outline

- Strings
- Indexing and slicing
- Errors and help function
- Variables
- Functions and loops

In [ ]:
```
x = 13.0
type(x)
```

In [ ]:
```
myname = "Evgenii"
type(myname)
```

We could index stings to access individual character:

In [ ]:
```
print("The first letter of my name is", myname[0], "and there are", len(myname), "letter
s in total.")
```

But strings are immutable:

In [ ]:
```
myname[0] = 'b'
```

Note that there are different printing options to display the same output:

In [ ]:
```
print("The first letter of my name is", myname[0], "and there are", len(myname), "letter
s in total.")
print(f"The first letter of my name is {myname[0]} and there are {len(myname)} letters i
n total.")
print("The first letter of my name is " + myname[0] + " and there are " + str(len(myname
)) + " letters in total.")
print("The first letter of my name is %s and there are %i letters in total." %(myname[0
], len(myname)))
```

Indexing and slicing

- Zero-based indexing
- Remember the input start:stop:step
- Excluding the stop
- Works same way for range

In [ ]:
```
mylist = ["one", "two", "three"]
mylist
```

In [ ]:
```
print("mylist[1] =", mylist[1])
print("mylist[2] =", mylist[2])
print("mylist[0] =", mylist[0])
print("mylist[-1] =", mylist[-1])
print("type(mylist[0]) =", type(mylist[0]))
```

In [ ]:
```
print("mylist[0:2] =", mylist[0:2])
print("mylist[:2] =", mylist[:2])
print("mylist[1:2] =", mylist[1:2])
print("type(mylist[1:2]) =", type(mylist[1:2]))
```

Keep in mind

- Practice and just try
- Zero-based indexing
- Final is excluded

Unlike strings, lists are mutable and there are many different methods available for lists as well:

In [ ]:
```
mylist[1] = 2
mylist.append('four')
mylist.insert(0, 'half')
mylist
```

In [ ]:
```
'three' in mylist
```

Errors and help function

- Python returns error statements when an error occurs
- Helps finding the mistake
- The following throws an error

In [ ]:
```
x = [3, 4, 7
```

In [ ]:
```
x.pop()
help(x.pop)
```

Variables

- Use sensible names: e.g. `ret` or `r` if they are returns, not `xyz_tmp`
- Do not use reserved words (keywords): e.g. return, def, for, False, lambda
- A variable name cannot start with a number
- Variable names are case_sensitive: `Ret` and `ret` are different variables
- Avoid variable names that are already used as build-in functions or methods: e.g. sum, int

In [ ]:
```
help("keywords")
```

In [ ]:
```
return = 3
```

In [ ]:
```
#sum = 3
type(sum)
```

In [ ]:
```
sum
#sum([1,2])
```

Functions

- Why do we need them?

Functions allow:

- to reuse pieces of code multiple times
- to break up a long code into smaller steps

- Note structure:
    - Start with def / for / while / if
    - Colon
    - Tabs / indentation
- Functions:
    - First define, then call
    - Namespace

In [ ]:
```
def function_name(input_parameters):

    # main body of the function

    return function_output
```

- What does the following function compute?
- Hint: `x` , `y` , and `z` must be between 1 and 10.

In [ ]:
```
def f(x,y,z):
    """Add docstring"""

    g = 0.20*x + 0.15*y + 0.65*z

    if z >= 5.0:
        g = round(g*2)/2  # Round to nearest half point
    else:
        g = min(g,5.0)  # Cannot be higher than 5.0

    return g
```

In [ ]:
```
f(9,8,9)
```

- Alternative implementation

In [ ]:

```
def f(x,y,z):
    """Computes the course grade."""

    g = 0.20*x + 0.15*y + 0.65*z

    g = round(g*2)/2  # Round to nearest half point

    if z < 5.0:
        g = min(g,5.0)  # Cannot be higher than 5.0

    return g
```

Functions descriptions could be specified using a one-line docstring:

In [ ]:
```
def SeeRange(n):
    """Returns a sequence of numbers in range(n)"""
    v = []
    for i in range(n):
        v.append(i)

    return v
```

In [ ]:
```
SeeRange(5)
```

With more complex functions it is advisable to write more elaborate multi-line docstring with summary, description of arguments, returns and exeptions. A docstring should give enough information to be able to run the function without reading the function's code. There are different conventions out there, but a beginner friendly one is Google docstring (Google Python Style Guide (https://google.github.io/styleguide/pyguide.html#383-functions-and-methods) & more examples (https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html)), that we extend here by including the type of arguments in description. For instance, a multi-line docstring for the funtion above might look as follows:

In [ ]:
```
def SeeRange(n):
    """Returns a sequence of numbers in range function

    Args:
        n (int): the end of a range

    Returns:
        v (list): numbers in range(n)
    """
    v = []
    for i in range(n):
        v.append(i)

    return v
```