CONTROL STRUCTURES AND FUNCTIONS

GENERAL BUSINESS

- Has everybody found a text editor that they like?
- Has everybody installed Anaconda?
- Can everybody open the Anaconda Command prompt?
- Who knows the basics of command prompt usage (changing director, listing directory contents)?

COMPARISON AND CONTROL STRUCTURES

BOOLEAN EXPRESSIONS

- Compare two objects and return `True` or `False`

```python
print(3 < 5) # less than
print(3 > 3) # greater than
print(type("text") == str) # equal
print("text" != 2) # not equal
```

```
True
False
True
True
```

IF CONDITION

- Checks if a boolean expression is True or False
- Executes code conditionally (only if the condition is satisfied)

```
if 3 < 5:
    print("Three is less than 5")

if 5 <= 3:
    print("Five is less or equal to three")
```

```
Three is less than 5
```

IF ELIF ELSE

- `if` alone is often not enough
- `elif` is short for `else if` and is checked if original if statement is not satisfied
- code under `else` is executed when if statement is not satisfied

```python
x = 1
if x > 1:
    print("Greater than one")
elif x < 1:
    print("Less than one")
else:
    print("Exactly one")
```

```
Exactly one
```

THE `IN` OPERATOR

Checks if an element is inside a list or dictionary

```
name = "Rick"
if name in ["Rick", "Morty"]:
    print("Name is in list")
```

```
Name is in list
```

```
key = 1
if key in {1: "one", 2: "two"}:
    print("key is in dict")

if key in {"one": 1, "two": 2}:
    print("Comparison of dicts is against keys by default")

if key in {"one": 1, "two": 2}.values():
    print("value is in dict")
```

```
key is in dict
value is in dict
```

THE `IS` OPERATOR

- The `is` operator checks the instance and not the values

```
var = None
one = 1
ls = [1, 2, 3]
print(var is None, var == None)
print(one is 1, one == 1)
print(ls is [1, 2, 3], ls == [1, 2, 3])
```

```
True True
True True
False True
```

THE `NOT` OPERATOR

- the `not` operator inverts a boolean expression

```
print(not False)
print(not 3 < 5)
```

```
True
False
```

BOOLEAN OPERATORS

The `and` and `or` operators can be used to combine multiple expressions

```
var = "test"
if 5 > 3 and var == "test":
    print("both statements are true")
```

```
both statements are true
```

```
if 5 > 3 or 8 > 2:
    print("at least one statement is true")
```

```
at least one statement is true
```

`PASS` AND `ASSERT`

- pass does nothing
- assert checks something and throws an Exception if not `True`

```
name = 2
assert type(name) == str, "name should be a string"
if type(name) != str:
    pass # can be useful when planning program structure
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: name should be a string
```

LOOPS

- A loop executes the same code a certain number of times.
- A loop can also execute the same code for each element in e.g. a list.

BASIC LOOP

```python
i = 0
data = [8, 4.5]

# the len function gets the length of a list
print("Length of the list is:", len(data))

while i < len(data):
    print(data[i])
    i += 1
```

```
Length of the list is: 2
8
4.5
```

Python can loop/iterate directly over the list elements

```
data = [ 1.73, 2.4122, 80, -4 ]

# iterate over elements, using keyword 'in'
for elem in data:
    print(elem)
```

```
1.73
2.4122
80
-4
```

ITERATE OVER MULTIPLE LISTS

```python
data = [ 1.73, 2.4122, 80, -4 ]
datanames = ["number 1", "number 2", "number 3", "number 4"]
for number, name in zip(data, datanames):
    print(name, number)
```

```
number 1 1.73
number 2 2.4122
number 3 80
number 4 -4
```

ITERATE OVER DICTIONARIES

```python
d = {"key1": 1, "key2": 2, "key3": 3}
for key in d:
    print(key)
```

```
key2
key1
key3
```

```python
d = {"key1": 1, "key2": 2, "key3": 3}
for key, item in d.iteritems():
    print(key, item)
```

SOMETIMES YOU NEED AN INDICES

range can be addressed with (start, stop, step)

```
print("range")
for i in range(1, 10, 2):
    print(i)
```

```
range
1
3
5
7
9
```

SOMETIMES ENUMERATE IS ALSO HANDY

```
l = ["a", "b", "c", "d"]
for i, item in enumerate(l):
    print(i, item)
```

```
0 a
1 b
2 c
3 d
```

IF/ELSE BLOCK

```python
dataset1 = [ 1.73, 2.4122, 80, -4 ]
# if/else blocks
for d in dataset1:
    if d > 3:
        res = ">3"
    else:
        res = "<=3"
    print(res)
```

```
<=3
<=3
>3
<=3
```

FOR LOOPS WITH BREAK AND ELSE

```python
for n in range(2, 8): # lets debug through this
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n / x)
            break # breaks out of (ends) current loop
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
```

FOR LOOPS AND CONTINUE

```
for num in range(2, 8):
    if num % 2 == 0: # percent sign is modulo
        print("Found an even number", num)
        continue # continue with the next iteration of the loop
    print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
```

MINI EXERCISE

```
####################################################################
# Mini Exercise 1

# Try to print only the positive numbers in 'dataset1',

####################################################################
dataset1 = [ 1.73, 2.4122, 80, -4 ]
```

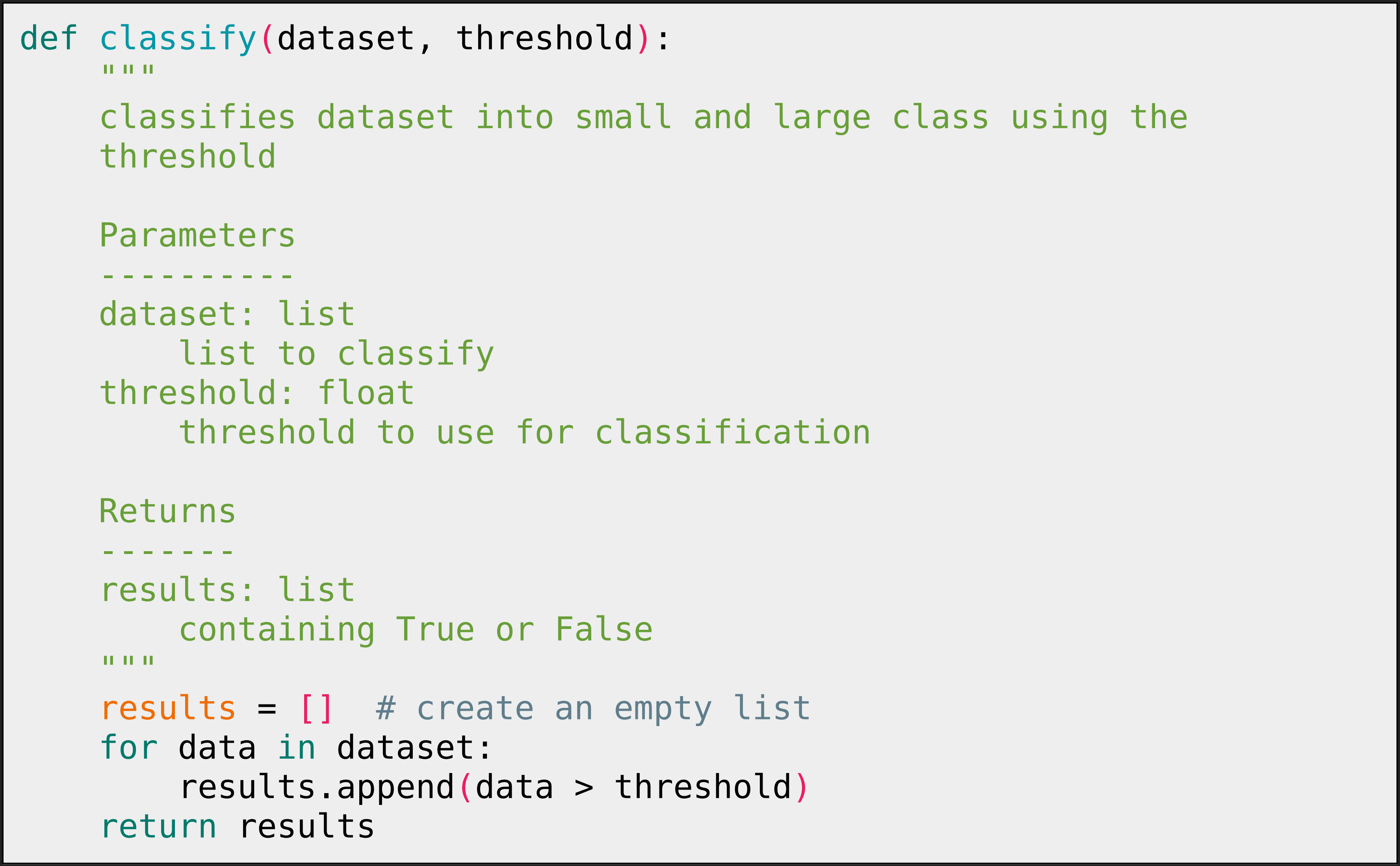
FUNCTIONS

- A function is a block of code that can be reused
- A function can take arguments and keywords
- A function can return a value
- It must be defined before we can use it

```
def function():
    print("print in a function")

function()
```

```
print in a function
```



```python
def classify(dataset, threshold):
    """
    classifies dataset into small and large class using the
    threshold

    Parameters
    ----------
    dataset: list
        list to classify
    threshold: float
        threshold to use for classification

    Returns
    -------
    results: list
        containing True or False
    """
    results = []  # create an empty list
    for data in dataset:
        results.append(data > threshold)
    return results
```

```
dataset1 = [1.73, 80, 2.4122, -4]
res = classify(dataset1, 2)
print(res)
print(classify(dataset1, 1))
```

```
>>> [False, True, True, False]
[True, True, True, False]
```

DOCUMENTATION

There are several ways to document code in Python.

Scientists often use the numpy Documentation style.

There is also PEP 257 Python Docstring Standard

Be Consistent

Documentation is useful for autocompletion in IDE

Generation of HTML or PDF Documentation using Sphinx.

FUNCTIONS WITH KEYWORD ARGUMENTS

```python
def classifydefault(dataset, threshold=2.5):
    """
    classifies dataset into small and large class using the
    threshold

    Parameters
    ----------
    dataset: list
        list to classify
    threshold: float, optional
        threshold to use for classification

    Returns
    -------
    results: list
        containing True or False
    """
    results = []  # create an empty list
    for data in dataset:
        results.append(data > threshold)
    return results
```

```
dataset2 = [1.73, 80, 2.4122, -4, 2.6]
print(classifydefault(dataset2))
```

```
[False, True, False, False, True]
```

NAMESPACES

Functions/Classes and Packages/Modules define their own local namespace.

```
variable = "I am Global"
var = "I'm also Global"
def func():
    variable = "I am Local"
    print(variable)
    print(var)

print(variable)
func()
```

```
I am Global
I am Local
I'm also Global
```

FUNCTIONS AND MUTABLE TYPES

Careful when changing a list in a function

```
l = [1, 2, 3]
def func(ls):
    ls.append(4)

print(l)
func(l)
print(l)
```

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

STRING FORMATTING

```
# String Formatting
# handy for any kind of logging, etc.
# mark replacement fields with curly braces
arg = 'world'
res = "hello {}".format(arg)
print(res)
```

```
hello world
```

```
res = "{} and {}".format("a pear", "a tree")
print(res)
```

```
a pear and a tree
```

```
# refer to arguments by index; possibly re-use them
res = "{0} and {1}, {1} and {0}".format("a pear", "a tree")
print(res)

# refer to arguments by name; possibly re-use them
res = "{good} is better than {bad}".format(good="some", bad="nothing")
print(res)
```

```
a pear and a tree, a tree and a pear
some is better than nothing
```

```python
# practically anything can be an argument to format(.)
value = 3.429188
res = "value is: {}".format(value)
print(res)

# custom formatting using format specifiers:
# format specifiers follow a colon inside the curly braces
# format as fixed point, with 3 digits after comma
res = "value is: {:.3f}".format(value)
print(res)
```

```
value is: 3.429188
value is: 3.429
```

```python
# format left-aligned, centered, and right-aligned
# with the given minimum width,
# and a trailing line-break
# prepare the template-string
tpl = "{:<15} {:^5} {:>10}\n"
# provide empty string to match all replacement fields
res = tpl.format("Carl Friedrich", "", "Gauss")
# re-use the template-string
res += tpl.format("Alexander", "von",  "Humboldt")
res += tpl.format("Gerhard", "", "Mercator")
print(res)
```

```
Carl Friedrich                Gauss
Alexander         von      Humboldt
Gerhard                    Mercator
```

```python
# multi-line strings can be formatted just as well.
res = """# This might be a {}-file-header,
# created by {}
# on {}""".format("text", "me", "2014-02-18")
print(res)
```

```
# This might be a text-file-header,
# created by me
# on 2014-02-18
```

Complete Format Specification Mini-Language:

http://docs.python.org/2/library/string.html#formatspec