Programming I

Michael Habeck
michael.habeck@uni-jena.de
Microscopic Image Analysis, University Hospital Jena

November 18, 2020

1 Image processing I - Programming I

In this lecture, we will learn the fundamentals of Python and start to look into writing our own functions.

Much of the material is inspired by J. VanderPlas: Whirlwind Tour of Python

1.1 Outline

- Recap of what we learned about Python so far (basic data types and operations)
- More on lists and other compound types, indexing and slicing
- User-defined functions
- Loops and control structures

1.2 Built-in data types

Python is a dynamic language and therefore doesn't require variable declarations specifying a variable's type (as, e.g., in C/C++). Simple scalar data types in Python are:

1.2.1 Scalar data types

Type	Example	Description
`int`	`x = 1`	integers (i.e., whole numbers)
`float`	`x = 1.0`	floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

1.2.2 Compound data types

Python also has several built-in compound types that act as containers for other types. These compound types are:

Type Name	Example	Description
`list`	`[1, 2, 3]`	Ordered collection
`tuple`	`(1, 2, 3)`	Immutable ordered collection
`dict`	`{'a':1, 'b':2, 'c':3}`	Unordered (key,value) mapping
`set`	`{1, 2, 3}`	Unordered collection of unique values

The following code produces examples for some of the built-in types provided by Python. Because Python is interpreted you can define these variables right on the spot:

[1]:
```
a = 1
b = 0.123456789
c = True
d = 'abcdefg'
e = [0, 'a', False, 0, 1.1]
f = (0, 'a', False, 0, 1.1)
g = set([0, 'a', False, 0, 1.1])
h = {None: [2,3], (1,2): 'xyz', 'image': 'processing'}
i = None

print('value of variable "a":', a)
print('value of variable "b":', b)
print('value of variable "c":', c)
print('value of variable "d":', d)
print('value of variable "e":', e)
print('value of variable "f":', f)
print('value of variable "g":', g)
print('value of variable "h":', h)
print('value of variable "i":', i)
```

```
value of variable "a": 1
value of variable "b": 0.123456789
value of variable "c": True
value of variable "d": abcdefg
value of variable "e": [0, 'a', False, 0, 1.1]
value of variable "f": (0, 'a', False, 0, 1.1)
value of variable "g": {0, 1.1, 'a'}
value of variable "h": {None: [2, 3], (1, 2): 'xyz', 'image': 'processing'}
value of variable "i": None
```

What is the `type` of each of these variables? Let's use Python to find out:

[2]:
```
print('type of variable "a":', type(a), a)
print('type of variable "b":', type(b), b)
print('type of variable "c":', type(c), c)
print('type of variable "d":', type(d), d)
print('type of variable "e":', type(e), e)
print('type of variable "f":', type(f), f)
```

```
print('type of variable "g":', type(g), g)
print('type of variable "h":', type(h), h)
print('type of variable "i":', type(i), i)
```

```
type of variable "a": <class 'int'> 1
type of variable "b": <class 'float'> 0.123456789
type of variable "c": <class 'bool'> True
type of variable "d": <class 'str'> abcdefg
type of variable "e": <class 'list'> [0, 'a', False, 0, 1.1]
type of variable "f": <class 'tuple'> (0, 'a', False, 0, 1.1)
type of variable "g": <class 'set'> {0, 1.1, 'a'}
type of variable "h": <class 'dict'> {None: [2, 3], (1, 2): 'xyz', 'image':
'processing'}
type of variable "i": <class 'NoneType'> None
```

Python integers don't overflow

[3]:
```
a = 2**200

print(a)
```

```
1606938044258990275541962092341162602522202993782792835301376
```

Floats are *double precision* floating point numbers. So they have a finite range:

[4]:
```
# floats

import sys

inf = float('inf')
print(1/inf)
floatmax = sys.float_info.max
floatmin = sys.float_info.min
print('largest float:', floatmax)
print('smallest float:', floatmin)
print('overflow:', floatmax * (1 + 2**(-52)))
# apparently, underflow occurs at numbers that are even smaller
# than floatmin
print('underflow:', 2**-1022, 2**-1074, 2**-1075)
```

```
0.0
largest float: 1.7976931348623157e+308
smallest float: 2.2250738585072014e-308
overflow: inf
underflow: 2.2250738585072014e-308 5e-324 0.0
```

1.2.3 Useful Built-In Functions

Type Name	Example	Description
`print`	`print('Hello, world!')`	Write to standard output
`type`	`type(1)`	Returns the type of an object
`len`	`len('abc')`	Size of object
`dir`	`dir(math)`	Content of module or object
`str`	`str(10)`	Convert object to string
`int`	`int('123')`	Convert string to integer
`range`	`range(1, 12)`	Creates a list

[5]:
```
# example code illustrating some built-in functions
import math
print(dir(math))
print(math.pi)
x = int('123')
print(type(x))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
3.141592653589793
<class 'int'>
```

1.3 Basic operations

1.3.1 Arithmetic operations

Python implements seven basic binary arithmetic operators, two of which can double as unary operators. They are summarized in the following table:

Operator	Name	Description
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference of `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	True division	Quotient of `a` and `b`
`a // b`	Floor division	Quotient of `a` and `b`, removing fractional parts
`a % b`	Modulus	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`
`+a`	Unary plus	`a` unchanged (rarely used)

These operators can be used and combined in intuitive ways, using standard parentheses to group operations. Numbers behave in the expected fashion under addition `+`, subtraction `-`, multiplication

* and division:

[6]:
```
# adding, subtracting, multiplying, dividing ints, floats and complex numbers

a, b, c = 123, 45678., 9+1.234j

print(a + b)
print(b - c)
print(a * c)
print(a / b)
```

```
45801.0
(45669-1.234j)
(1107+151.782j)
0.0026927623801392356
```

The operator `//` is the floor division:

[7]:
```
# division of two ints can yield a float
# floor division of two ints yields an int
print(10 / 7)
print(10 // 7)
print(10 % 7)
print(10 - 7 * (10//7))
print(10**3)

# floor division of two floats yields a float
print(10 / 7.)
print(10 // 7.)
```

```
1.4285714285714286
1
3
3
1000
1.4285714285714286
1.0
```

1.3.2 Comparison Operations

Another type of operation which can be very useful is comparison of different values. For this, Python implements standard comparison operators, which return Boolean values `True` and `False`. The comparison operations are listed in the following table:

Operation	Description
`a == b`	`a` equal to `b`
`a != b`	`a` not equal to `b`
`a < b`	`a` less than `b`
`a > b`	`a` greater than `b`

Operation	Description
`a <= b`	`a` less than or equal to `b`
`a >= b`	`a` greater than or equal to `b`

[8]:
```
print('ab' == 'ab')
print(1 == 1+1)
print(2 != 1+1)
print(6-1 <= 5)
print('a' < 'b')
print('a' > 'b')
```

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

1.3.3 Logical operations

Python provides operators to combine Boolean values using the keywords `and`, `or`, and `not`:

Operation	Name	Example
`a and b`	AND	`(x < 6) and (x > 2)` is true for `x = 4`
`a or b`	OR	`(x > 10) or (x % 2 == 0)` is also true
`not a`	NOT	`not (x < 6)` is false

There is no special XOR operator, but a simple version of XOR is to use the operator `=!` as in ~~~ python (x > 1) != (x < 10) ~~~

[9]:
```
A = True
B = 10 < 7
print(A and B)
print(A or B)
print(not B)
```

```
False
True
True
```

1.3.4 Identity and Membership Operators

Like `and`, `or`, and `not`, Python also contains prose-like operators to check for identity and membership. They are the following:

Operator	Description
`a is b`	True if `a` and `b` are identical objects
`a is not b`	True if `a` and `b` are not identical objects
`a in b`	True if `a` is a member of `b`
`a not in b`	True if `a` is not a member of `b`

[10]:
```
a = 10
b = a
print(a is b)
b = 10
print(a is b)
a = [1, 2]
print(0 in a)
```

```
True
True
False
```

1.3.5 Bitwise operations

In addition to the standard numerical operations, Python includes operators to perform bitwise logical operations on integers. These are much less commonly used than the standard arithmetic operations, but it's useful to know that they exist. The six bitwise operators are summarized in the following table:

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both `a` and `b`
a | b	Bitwise OR	Bits defined in `a` or `b` or both
`a ^ b`	Bitwise XOR	Bits defined in `a` or `b` but not both
`a << b`	Bit shift left	Shift bits of `a` left by `b` units
`a >> b`	Bit shift right	Shift bits of `a` right by `b` units
`~a`	Bitwise NOT	Bitwise negation of `a`

[11]:
```
# (14)_10 : 01110
# (22)_10 : 10110
# ===============
#   AND   : 00110 -> (6)_10
#   OR    : 11110 -> (30)_10
#  XOR    : 11000 -> (24)_10
print('AND:', 14 & 22)
print(' OR:', 14 | 22)
print('XOR:', 14 ^ 22)

print('\nLeft/right shift')
print(5<<1) # (5)_10 -> (10)_10 since 101 -> 1010
```

```
print(5>>2) # (5)_10 -> (2)_10 since 101 -> 10

print('\nBitwise NOT')
print(~5)   # (5)_10 = 0101 -> 1010 (6)_10
print(~-6)  # inversion
print(~~5)
print(~7)   # (7)_10 = 0111 -> 1000 (8)_10
print(~-8)  # inversion
```

```
AND: 6
 OR: 30
XOR: 24

Left/right shift
10
1

Bitwise NOT
-6
5
5
-8
7
```

1.4 Lists, tuples, sets

[12]:
```
# various ways of creating lists

a = [1, 3, 'a', 'bc']
b = list(range(3, 5, 2))
c = list('abcdef')
d = [x**2 for x in range(5)]

print(a)
print(b)
print(c)
print(d)
```

```
[1, 3, 'a', 'bc']
[3]
['a', 'b', 'c', 'd', 'e', 'f']
[0, 1, 4, 9, 16]
```

[13]:
```
# a list of lists
x = [[1,2,3],[4,5,6],[7,8,9], 'A']
print(x)
print(x[1])
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], 'A']
[4, 5, 6]
```

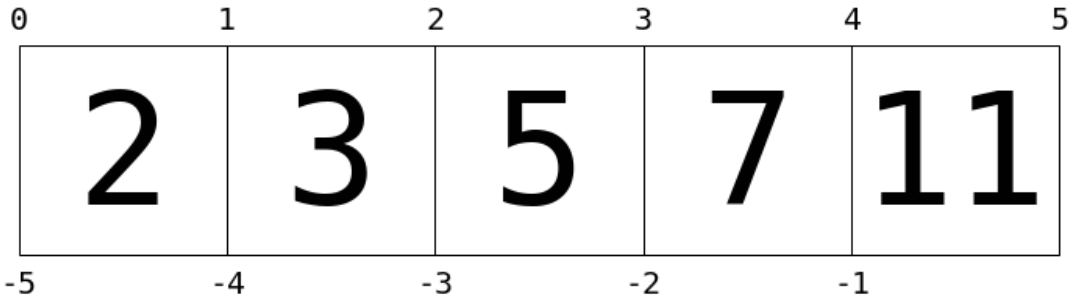
1.4.1 Indexing and slicing

Python provides access to elements in compound types through **indexing** for single elements, and **slicing** for multiple elements. Python uses zero-based indexing, so we can access the first and second element in a list `l` using the following syntax `l[0]` and `l[1]`. Elements at the end of the list can be accessed with negative numbers starting from -1: `l[-1]` and `l[-2]` return the last and the second last element of `l`:

[14]:
```
l = [2, 3, 5, 7, 11]
print('list: ', l)
print('1st element:', l[0])
print('2nd element:', l[1])
print('last element:', l[-1], l[len(l)-1]) # see MATLAB a(end)
print('2nd last element:', l[-2])
```

```
list:  [2, 3, 5, 7, 11]
1st element: 2
2nd element: 3
last element: 11 11
2nd last element: 7
```

This figure illustrates indexing for this particular example (source: J. VanderPlas: Whirlwind Tour of Python)



Slicing can be used to access sublists. Slices have the following structure `l[start:stop:step]` where `start`, `stop`, and `step` can be integers or `None`. `start` specifies where the slice starts, `stop` where it ends; `step` is an increment that allows us to skip elements in the slice.

[15]:
```
# accessing sublists in l
print(l)

# every second element in list 'l' start with the first one
print(l[:-1:2])
```

```
# every second element in list 'l' start with the 2nd one
print(l[1::2])

# reversed list
print(l[::-1])
```

```
[2, 3, 5, 7, 11]
[2, 5]
[3, 7]
[11, 7, 5, 3, 2]
```

1.4.2 Mutable versus immutable containers

Lists are mutable, i.e. their content can be changed:

[16]:
```
a = [1, 3, 'a', 'b']
b = list(range(5))
c = list('abcdef')
d = [x**2 for x in range(5)]

print('before')
print(a)
print(b)
print(c)
print(d)

a[0] = 10
b.append(10)
c.extend(d)
#c = c + d

print('\nafter')
print(a)
print(b)
print(c)
```

```
before
[1, 3, 'a', 'b']
[0, 1, 2, 3, 4]
['a', 'b', 'c', 'd', 'e', 'f']
[0, 1, 4, 9, 16]

after
[10, 3, 'a', 'b']
[0, 1, 2, 3, 4, 10]
['a', 'b', 'c', 'd', 'e', 'f', 0, 1, 4, 9, 16]
```

[17]:
```
# list methods
print(dir(a))

b.insert(2, 23452435)
print(b)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
[0, 1, 23452435, 2, 3, 4, 10]
```

Tuples are immutable, meaning their content cannot be changed:

[18]:
```
# tuples

a = (1, 2, 3)
b = 1, 2, 3
c = tuple('abc')

print(a, type(a))
print(b, type(b))
print(c, type(c))

print(a + b + c)
```

```
(1, 2, 3) <class 'tuple'>
(1, 2, 3) <class 'tuple'>
('a', 'b', 'c') <class 'tuple'>
(1, 2, 3, 1, 2, 3, 'a', 'b', 'c')
```

[19]:
```
a = (1, 2, 3)
a[0] = 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-fa208dcb9091> in <module>
      1 a = (1, 2, 3)
----> 2 a[0] = 'a'

TypeError: 'tuple' object does not support item assignment
```

[20]:
```
a = list(a)
a[0] = 'a'
print(a)
```

```
['a', 2, 3]
```

1.5 User-defined functions

Functions allow you to package reusable pieces of code and use the functionality over and over again without reproducing the code. This helps us to develop compact, bug-free and reusable code. The Python keyword `def` marks the definition of a function. This keyword is followed by the function name and input variables that are listed inside round brackets (this list can also be empty); a colon marks the end of the function header:

```
def function_name(variables1, variable2, ...):
    """
    Optional doc string
    """
    pass
```

The preferred convention for function names is to use snake_case, i.e. lower case words separated by an underscore. (See the PEP8 guidelines for Python programming for more details on how to write good Python code.)

Right below the first line, we can place an optional documentation string (*doc string*) followed by the main function code, both indented to the right by a tab. It is not mandatory, but good programming practice to briefly describe what the function does, what input it expects and what the output will be.

```
def function_name(variables1, variable2, ...):
    """
    Optional doc string
    """
    # main body of the function: set of operations to be performed
    # by the function
    return result
```

Here the main body of the code is commented out (remember that `#` marks a comment which extends until the end of the line). The keyword `return` is used to return the function's output (if not implemented the function returns `None`). Useful functions can be packaged in a *module* whose filename should end with the .py extension.

The following function computes the surface area and volume of a cylinder:

[21]:
```
def cylinder(radius, height):
    """
    Computes the surface and volume of a cylinder
    """
    from math import pi

    surface = 2 * pi * radius * height
```

```
    volume = pi * radius**2 * height

    return surface, volume
```

Some comments:

- we need to fetch the value of π from the built-in module `math`
- the exponentiation operator is double asteriks `**` so `3**4` is 81 (see section "Arithmetic operations")
- the function returns a tuple of two values

We can call this function with some inputs and store the output as follows:

[22]:
```
result = cylinder(1, 2)
print(result)
```

```
(12.566370614359172, 6.283185307179586)
```

If we know that the function returns a tuple of two values, we can also directly assign them to specific variables (of course the variable names can be different from the ones used internally in the implementation of `cylinder`):

[23]:
```
surf, vol = cylinder(1, 2)
print(f'the surface area is {surf}')
print(f'the volume is {vol}')
```

```
the surface area is 12.566370614359172
the volume is 6.283185307179586
```

Python is not typed, therefore the input variables could be any quantity. If the input variables implement the operations that are executed in the function body (such as multiplication and exponentation), the function will also produce a useful output. For example, we could use numpy-arrays as inputs (much more about numpy later):

[24]:
```
# this doesn't work since the input doesn't support
# the arithmetic operations carried out in 'cylinder'
result = cylinder('radius', 'height')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-53f5ee921d0a> in <module>
      1 # this doesn't work since the input doesn't support
      2 # the arithmetic operations carried out in 'cylinder'
----> 3 result = cylinder('radius', 'height')

<ipython-input-21-5bb0b10773d4> in cylinder(radius, height)
      5     from math import pi
      6
----> 7     surface = 2 * pi * radius * height
      8     volume = pi * radius**2 * height
```

```
      9

TypeError: can't multiply sequence by non-int of type 'float'
```

[25]:
```
import numpy as np

result = cylinder(np.arange(1,6), np.arange(1,6))
print(f'surface: {result[0]}')
print(f' volume: {result[1]}')
```

```
surface: [  6.28318531  25.13274123  56.54866776 100.53096491 157.07963268]
 volume: [  3.14159265  25.13274123  84.82300165 201.06192983 392.6990817 ]
```

This function call returns a 2-tuple of arrays where each array has five values corresponding to the five values in the input arrays specifying the radii and heights.

We can also call other user-defined functions within a function:

[26]:
```
def circle(radius):
    """
    Computes the circumference and area of a circle
    """
    from math import pi

    circ = 2 * pi * radius
    area = pi * radius**2

    return circ, area


def cylinder(radius, height):
    """
    Computes the surface and volume of a cylinder
    """
    circ, area = circle(radius)

    return circ * height, area * height


print(cylinder(1, 2))
```

```
(12.566370614359172, 6.283185307179586)
```

We can also define functions within a function and call them:

[27]:
```
def cylinder(radius, height):
    """
    Computes the surface and volume of a cylinder
    """
```

```
    def circle2(radius):
        """
        Computes the circumference and area of a circle
        """
        from math import pi

        circ = 2 * pi * radius
        area = pi * radius**2

        return circ, area

    circ, area = circle2(radius)

    return circ * height, area * height

print(cylinder(1, 2))
```

```
(12.566370614359172, 6.283185307179586)
```

The function `circle2` is only known locally within the *namespace* of the function `cylinder`. It is unknown outside the function `cylinder`:

[28]:
```
# 'circle' can be called because it was defined before
# in the global name space
print(circle(1.))

# 'circle2' cannot be called because it was defined only
# within 'cylinder'
print(circle2(1.))
```

```
(6.283185307179586, 3.141592653589793)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-36687c049065> in <module>
      5 # 'circle2' cannot be called because it was defined only
      6 # within 'cylinder'
----> 7 print(circle2(1.))

NameError: name 'circle2' is not defined
```

1.5.1 Namespaces: local versus global variables

The variables that are created inside a function are local variables, they only exist temporarily and are not accessible from outside: they are not part of the global namespace, but only of the local namespace defined within a function. All local variables are moved to the garbage collector (i.e. they are deleted in the long run) unless they are returned.

[29]:
```
def f(x):
    """
    Illustrating local and global variables
    """
    # local variable that will not be accessible outside
    y = x**2

    # local variable that will be accessible outside since it is returned
    z = y + 1

    # show object's identity to check if indeed the local variable and
    # the one generated by calling the function point to the same memory
    # note that 'id(x)' returns the (unique) identity of python object 'x'
    print('identity of local variable "z" : {}'.format(id(z)))

    return z

x = 10
a = f(x)
print('identity of global variable "a": {}'.format(id(a)))
print('Is a variable named "y" part of the global namespace? - {}'.format(
    'y' in globals()))
print('Is a variable named "a" part of the global namespace? - {}'.format(
    'a' in globals()))
```

```
identity of local variable "z" : 93873154654080
identity of global variable "a": 93873154654080
Is a variable named "y" part of the global namespace? - False
Is a variable named "a" part of the global namespace? - True
```

Since the identity of the objects to which the local variable `z` and the global variable `a` refer is identical, they address the same memory. All variables that are part of the global namespace can be listed by calling the built-in function

```
globals()
```

It is also possible to define global variables with the Python keyword `global`, but since the use of global variables is not recommended, we won't discuss global variables any further.

1.6 Algorithms and control structures

An *algorithm* is an ordered sequence of precisely defined instructions that performs some task in a finite amount of time. Ordered means that the instructions can be numbered, but an algorithm must have the ability to alter the order of its instructions using a control structure. There are three categories of algorithmic operations:

1. **Sequential operations**: Instructions executed in order.
2. **Conditional operations**: Control structures that first ask a question to be answered with a true/false answer and then select the next instruction based on the answer.

3. **Iterative operations (loops)**: Control structures that repeat the execution of a block of instructions.

1.6.1 Control structures: the `if`, `elif` and `else` keywords

The `if` statement's basic form is

```
if condition:
    # do something
    pass
```

Every `if` statement terminates with a colon and must be followed by a block of commands that is indented to the right and will be executed, if the condition is met. A condition is a boolean-valued expression such as `i < 10`.

[30]:
```
a = 10
if a < 20:
    print(f'value of "a" inside if block: {a}')
    a = 20
print(f'value of "a" outside if block: {a}')
```

```
value of "a" inside if block: 10
value of "a" outside if block: 20
```

The basic structure for the use of the `if-else` statement is

```
if condition:
    # do something
    pass
else:
    # do something else
    pass
```

The general form of the `if` statement is

```
if condition1:
    # statement1
    pass
elif condition2:
    # statement2
    pass
else:
    # statement3
    pass
```

The `else` and `elif` keywords may be omitted if not required. However, if both are used, the `else` statement must come after the `elif` statement to take care of all conditions that are not met.

Example:

[31]:
```
def f(value):
    if value == 0:
```

```
        print('value is 0')
    elif value == 1:
        print('value is 1')
    elif value == -1:
        print('value is -1')
    else:
        print('value is neither -1, 0, 1')

f(0)
f(1)
f(-1)
f(100)
```

```
value is 0
value is 1
value is -1
value is neither -1, 0, 1
```

[32]:
```
# nested conditions
a = 10
if a < 11:
    if a > 9:
        print(a)
    else:
        print(a**2)
```

```
10
```

1.7 Loops

1.7.1 The `for` loop

A simple example of a `for` loop is

```
for k in range(0, 11, 2):
    # for loop block
    pass
```

The loop variable `k` is initially assigned the value 0, each successive pass through the loop increments `k` by 2 until the value 10 is reached. The program then continues to execute any statements following the end statement.

[33]:
```
for k in range(0, 11, 2):
    print(k, end=' ')
```

```
0 2 4 6 8 10
```

The `for` loop can loop over any iterable quantity such as a list or a tuple. For example:

[34]:
```
# create a list
a = ['x', 'y', 0, 10]

# loop over list and print elements
for elem in a:
    print(elem, end=' ')
```

```
x y 0 10
```

A convenient built-in command is

`enumerate`

which returns an iterator that yields 2-tuples where the first element of the tuple is the list index and the second element is the list element:

[35]:
```
for i, elem in enumerate(a):
    print(f'The {i+1}-th element of list "a" is {elem} (but the index is {i})')
```

```
The 1-th element of list "a" is x (but the index is 0)
The 2-th element of list "a" is y (but the index is 1)
The 3-th element of list "a" is 0 (but the index is 2)
The 4-th element of list "a" is 10 (but the index is 3)
```

Another useful python command is

`zip`

It allows you to zip up multiple lists in a zipper-like fashion:

[36]:
```
# create three lists to be zipped
a = ['x', 'y', 0, 10]
b = [-1, -10, None, 'q']
c = ['a', 'b', 'c']

for t in zip(a, b, c):
    print(t)
```

```
('x', -1, 'a')
('y', -10, 'b')
(0, None, 'c')
```

Note that the shortest list defines the number of tuples that are yielded by `zip`. In the above example, since list `c` has only three elements (whereas lists `a`, `b` have four), the iterator created by the `zip` command yields also only three 3-tuples.

1.7.2 The `while` loop

The `while` loop is used when the looping process terminates because a specified condition is satisfied, and thus the number of passes is not known in advance. A simple example of a while loop is

```
while condition:
    # while code block
    pass
```

For the correct execution of a while loop the following two conditions must be satisfied:

- The loop variable must have a value before the `while` statement is executed
- The loop variable must be changed somehow by the statements

Here is a simple example of a `while` loop:

[37]:
```
x = 5
while x < 25:
    print(x, end=' ')
    x = 2*x - 1
```

```
5 9 17
```

The Python keyword `break` allows you to break out of a while loop.

[38]:
```
# without the break statement this loop would run
# forever
x = 5
while True:
    print(x, end=' ')
    x = 2*x - 1
    if x > 12:
        break
```

```
5 9
```

A little quiz: Before executing the code try to guess what will be printed to standard output:

[39]:
```
a, b = 1, 1

while a < 100:
    print(a, end=' ')
    a, b = b, a+b
```

```
1 1 2 3 5 8 13 21 34 55 89
```

1.8 General structure of a Python script

The general structure of a Python program or script is as follows:

```
"""
Documentation briefly explaining the script's functionality
"""
# load modules that are needed by the script
import something

# User-defined functions
```

```
def my_function1():
    pass

# Test code or main code
if __name__ == '__main__':
    # Put some code running the above function(s) here
    # For example
    my_function1()
```

By using the condition

```
if __name__ == '__main__':
```

you can at the same time

1. **test** your code while developing it,
2. **use** the functions that you defined yourself in some other code **without** executing the test code.

A simple example is the following script `circle.py`

[40]:
```
"""
This module defines a constant 'twopi' and computes the circumference
of a circle
"""
import math

# define 2*pi
twopi = 2 * math.pi

def circumference(radius):
    """
    Computes the circumference of a circle with the given radius
    """
    return twopi * radius

if __name__ == '__main__':
    # some test code
    print(circumference(1.))
```

```
6.283185307179586
```

You could save this piece of code in a py-file called, e.g., `circle.py` and use it in some other other "client" code:

```
# import my own module (without executing the test code)
import circle
l = circle.circumference(2.)
print(circle.twopi)
```

1.9 Strings

1.9.1 String methods

[41]:
```
s = 'Image Processing I'
print(s)
print(s.split())
print(s.lower())
print(s.replace(' ', '+'))
print(s.count('s'))
print(s.index('g'))
```

```
Image Processing I
['Image', 'Processing', 'I']
image processing i
Image+Processing+I
2
3
```

1.9.2 Creating strings with format strings

[42]:
```
s = '{0:d}'
print(s.format(12))
s = '{0:08b}'
print(s.format(45))
```

```
12
00101101
```