Preparatory Course Informatics for Life Scientists

An Introduction to Python 2:
Data Types and Comparison

Philipp Thiel

September 13, 2022

Data Types

- In programming, **data type** (often just type) is an important concept
- Computer memory can just store 0 and 1 (sure you know ;)
- Programming languages need to know how to interpret these bits

Data Types

- In programming, **data type** (often just type) is an important concept
- Computer memory can just store 0 and 1 (sure you know ;)
- Programming languages need to know how to interpret these bits

- Python comes with a set of **built-in** data types
- These types can be grouped into categories

Text	Numeric	Boolean	Sequence	Mapping	Set	Binary	None
str	int	bool	list	dict	set	bytes	NoneType
	float		tuple		fozenset	bytearray	
	complex		range			memoryview	

Data Types

- A data type has a **set of possible values**
- A data type has a **set of operations**

- Python is a so-called **dynamically typed** language
- That is, we do not need to specify what data type a variable stores
- Misuse of a date type usually leads to a **TypeError**

- The **type()** function returns the type of a variable
- This function is again a built-in function

- In fact: data types are modelled using **classes**!
- Thus: objects (class instances) are of a certain type!

Data Types: str

- We already met that type: strings are **sequences of characters**
- Explicitly specified in source code by string literals (in quotes)
- Many functions return read data as type str
- The set of possible values includes almost all characters
- We already have seen some operations (operators): '+', '*', 'in'
- However, many more operations are possible on type str

Data Types: str

- We already met that type: strings are **sequences of characters**
- Explicitly specified in source code by string literals (in quotes)
- Many functions return read data as type str
- The set of possible values includes almost all characters
- We already have seen some operations (operators): '+', '*', 'in'
- However, many more operations are possible on type str

myscript.py

```
1  var_1 = "This is a variable of type str"
2  print( type(var_1) )
```

Data Types: str

- We already met that type: strings are **sequences of characters**
- Explicitly specified in source code by string literals (in quotes)
- Many functions return read data as type str
- The set of possible values includes almost all characters
- We already have seen some operations (operators): '+', '*', 'in'
- However, many more operations are possible on type str

myscript.py

```
1  var_1 = "This is a variable of type str"
2  print( type(var_1) )
```

```
1  $ python myscript.py
2  <class 'str'>
3  $
```

Lesson: variable 'var_1' is of type / has type 'str'

Data Types: str

myscript.py

```
1  var_1 = "Charly"
2
3  x = len(var_1)
4
5  print( x )
6  print( type(x) )
```

Data Types: str

myscript.py

```
1  var_1 = "Charly"
2
3  x = len(var_1)
4
5  print( x )
6  print( type(x) )
```

```
1  $ python myscript.py
2  30
3  <class 'int'>
4  $
```

Lessons:

- Again a built-in function: **len()** returns the length of a sequence
- Thus, the return type of this function is an integer number

Data Types: int

- The integer type represents **integer numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, ’+’, and ’-’

Data Types: int

- The integer type represents **integer numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, '+', and '-'

myscript.py

```
1  x = 12
2  y = +12
3  z = -12
4  print( type(x) )
5  print( type(y) )
6  print( type(z) )
```

Data Types: int

- The integer type represents **integer numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, '+', and '-'

myscript.py

```
1  x = 12
2  y = +12
3  z = -12
4  print( type(x) )
5  print( type(y) )
6  print( type(z) )
```

```
1  $ python myscript.py
2  <class 'int'>
3  <class 'int'>
4  <class 'int'>
5  $
```

Data Types: int

myscript.py

```
1  x = 12
2  y = 5
3
4  print( x + y )
5  print( x - y )
6  print( x * y )
7  print( x % y )
8  print( x ** y )
```

Data Types: int

myscript.py

```
1  x = 12
2  y = 5
3
4  print( x + y )
5  print( x - y )
6  print( x * y )
7  print( x % y )
8  print( x ** y )
```

```
1  $ python myscript.py
2  17
3  7
4  60
5  2
6  248832
7  $
```

Lesson: the modulo operator '%' calculates b from equation $x = m * y + b$

Data Types: int

myscript.py

```
1  y = -12
2  print(y)
3  print( abs(y) )
4
5  x = 12 / 3
6  print( x )
7  print( type(x) )
```

Data Types: int

myscript.py

```
1  y = -12
2  print(y)
3  print( abs(y) )
4
5  x = 12 / 3
6  print( x )
7  print( type(x) )
```

```
1  -12
2  12
3  4.0
4  <class 'float'>
```

Lessons:

- Again a built-in function: **abs()** returns the absolute value
- Operations can have a return type different from the operand types

Data Types: float

- The floating point type represents **real numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, '+', '-', and '.'
- The int operations we met are also available for float

Data Types: float

- The floating point type represents **real numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, '+', '-', and '.'
- The int operations we met are also available for float

myscript.py

```
1  x = 3.1
2  y = 2
3
4  print( type(x) )
5  print( x * y)
```

Data Types: float

- The floating point type represents **real numbers**
- In source code integers are written without quotes
- Set of possible values: numeric characters, '+', '-', and '.'
- The int operations we met are also available for float

myscript.py

```
1  x = 3.1
2  y = 2
3
4  print( type(x) )
5  print( x * y)
```

```
1  $ python myscript.py
2  <class 'float'>
3  6.2
4  $
```

Data Types: float

myscript.py

```
1  x = 3.14159265358979323846264338327950288419 71
2
3  print( round(x) )
4  print( round(x, 6) )
```

Data Types: float

myscript.py

```
1  x = 3.14159265358979323846264338327950288841971
2
3  print( round(x) )
4  print( round(x, 6) )
```

```
1  $ python myscript.py
2  3
3  3.141593
4  $
```

Lessons:

- Again a built-in function: **round()**
- Functions can have more than one **argument** (also no argument)

Data Types: list

- List is a sequence to store multiple objects (often called **elements**)
- Lists are **mutable** (in contrast to the closely related type **tuple**)
- The order of inserted objects is preserved

Data Types: list

- List is a sequence to store multiple objects (often called **elements**)
- Lists are **mutable** (in contrast to the closely related type **tuple**)
- The order of inserted objects is preserved

myscript.py

```
1  lst0 = []
2  lst1 = [1, 2, 3, 4]
3
4  print(lst0)
5  print(lst1)
6  print( type(lst0) )
```

```
1  $ python myscript.py
```

Data Types: list

- List is a sequence to store multiple objects (often called **elements**)
- Lists are **mutable** (in contrast to the closely related type **tuple**)
- The order of inserted objects is preserved

myscript.py

```
1  lst0 = []
2  lst1 = [1, 2, 3, 4]
3
4  print(lst0)
5  print(lst1)
6  print( type(lst0) )
```

```
1  $ python myscript.py
2  []
3  [1, 2, 3, 4]
4  <class 'list'>
5  $
```

Data Types: list

myscript.py

```
1  lst0 = []
2  lst1 = [1, 2, 3, 4]
3
4  print( len(lst0) )
5  print( len(lst1) )
```

```
1  $ python myscript.py
```

Data Types: list

myscript.py

```
1  lst0 = []
2  lst1 = [1, 2, 3, 4]
3
4  print( len(lst0) )
5  print( len(lst1) )
```

```
1  $ python myscript.py
2  0
3  4
4  $
```

Lesson: the **len()** function also works for lists (which is a sequence)

Data Types: list

myscript.py

```
1  lst1 = [1, 2, 3, 4]
2  lst2 = ["a", "b", "c", "d"]
3
4  print(lst1 + lst2)
5  print(lst2 + lst1)
```

```
1  $ python myscript.py
```

Data Types: list

myscript.py

```
1  lst1 = [1, 2, 3, 4]
2  lst2 = ["a", "b", "c", "d"]
3
4  print(lst1 + lst2)
5  print(lst2 + lst1)
```

```
1  $ python myscript.py
2  [1, 2, 3, 4, 'a', 'b', 'c', 'd']
3  ['a', 'b', 'c', 'd', 1, 2, 3, 4]
4  $
```

Lesson: lists can store objects of different type

Data Types: list

- Elements can be accessed via their **index** (position) in the list

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[1])
4  print( lst[0])
5  print( lst[3])
6  print( lst[2])
```

```
1  $ python myscript.py
```

Data Types: list

- Elements can be accessed via their **index** (position) in the list

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[1])
4  print( lst[0])
5  print( lst[3])
6  print( lst[2])
```

```
1  $ python myscript.py
2  b
3  a
4  d
5  c
6  $
```

Lesson: programmers start counting at 0!

Data Types: list

- Elements can be accessed via their **index** (position) in the list

myscript.py

```
lst = ["a", "b", "c", "d"]

print( lst[-1])
print( lst[-2])
print( lst[-3])
print( lst[-4])
```

```
$ python myscript.py
```

Data Types: list

- Elements can be accessed via their **index** (position) in the list

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[-1])
4  print( lst[-2])
5  print( lst[-3])
6  print( lst[-4])
```

```
1  $ python myscript.py
2  d
3  c
4  b
5  a
6  $
```

Lesson: negative indices count from the end of the list

Data Types: list

- Element ranges can be accessed via so-called **slicing**

myscript.py

```
lst = ["a", "b", "c", "d"]

print( lst[0:1])
print( lst[0:2])
print( lst[0:3])
print( lst[0:4])
```

```
$ python myscript.py
```

Data Types: list

- Element ranges can be accessed via so-called **slicing**

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[0:1])
4  print( lst[0:2])
5  print( lst[0:3])
6  print( lst[0:4])
```

```
1  $ python myscript.py
2  ['a']
3  ['a', 'b']
4  ['a', 'b', 'c']
5  ['a', 'b', 'c', 'd']
6  $
```

Lesson: The slice corresponds to the interval [start, end[

Data Types: list

- Element ranges can be accessed via so-called **slicing**

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[-4:-1])
```

```
1  $ python myscript.py
```

Data Types: list

- Element ranges can be accessed via so-called **slicing**

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  print( lst[-4:-1])
```

```
1  $ python myscript.py
2  ['a', 'b', 'c']
3  $
```

Lesson: Slicing also works with negative indices

INTERMEZZO: Objects Revisited

- Objects have **properties** and **methods**
- These (class) methods confer functionality to its instances
- They are called like a function but via a concrete object

myscript.py

```
1  lst = ["a", "b", "c", "d"]
2
3  # Call append() on the object lst to append string object 'e'
4  lst.append("e")
5
6  print(lst)
```

```
1  $ python myscript.py
```

INTERMEZZO: Objects Revisited

- Objects have **properties** and **methods**
- These (class) methods confer functionality to its instances
- They are called like a function but via a concrete object

myscript.py

```
lst = ["a", "b", "c", "d"]

# Call append() on the object lst to append string object 'e'
lst.append("e")

print(lst)
```

```
$ python myscript.py
['a', 'b', 'c', 'd', 'e']
$
```

Lessons:

- The method **append** exists for type str
- This method extends the given list by appending it's argument

Data Types: list

Type list

myscript.py

```
1  lst = [1, 3, 4, 5]
2  lst.insert(1, 2)
3
4  lst.reverse()
5  print(lst)
6
7  lst.clear()
8  print(lst)
```

```
1  $ python myscript.py
```

Data Types: list

Type list

myscript.py

```
1  lst = [1, 3, 4, 5]
2  lst.insert(1, 2)
3
4  lst.reverse()
5  print(lst)
6
7  lst.clear()
8  print(lst)
```

```
1  $ python myscript.py
2  [5, 4, 3, 2, 1]
3  []
4  $
```

Data Types: list

Type list

myscript.py

```
1  lst1 = [1, 3, 4, 5]
2  lst2 = [-1, 0]
3
4  lst3 = lst1.copy()
5  lst4 = lst2.copy()
6
7  lst2.extend(lst1)
8  print(lst2)
9
10 lst4 += lst3
11 print(lst4)
```

```
1  $ python myscript.py
```

Data Types: list

Type list

myscript.py

```
1  lst1 = [1, 3, 4, 5]
2  lst2 = [-1, 0]
3
4  lst3 = lst1.copy()
5  lst4 = lst2.copy()
6
7  lst2.extend(lst1)
8  print(lst2)
9
10 lst4 += lst3
11 print(lst4)
```

```
1  $ python myscript.py
2  [-1, 0, 1, 3, 4, 5]
3  [-1, 0, 1, 3, 4, 5]
4  $
```

Data Types: list

Type list

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[:])
4  print( lst[:1])
5  print( lst[3:])
```

```
1  $ python myscript.py
```

Data Types: list

Type **list**

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[:])
4  print( lst[:1])
5  print( lst[3:])
```

```
1  $ python myscript.py
2  [1, 3, 4, 5]
3  [1]
4  [5]
5  $
```

Lessons:

- Leaving <start_index> empty starts the slice at the first element
- Leaving <end_index> empty extends the slice up to the last element

Data Types: list

Type **list**

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[3] )
4  print( lst[4] )
```

```
1  $ python myscript.py
```

Data Types: list

Type **list**

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[3] )
4  print( lst[4] )
```

```
1  $ python myscript.py
2  5
3  Traceback (most recent call last):
4    File "myscript.py", line 4, in <module>
5      print( lst[4] )
6  IndexError: list index out of range
7  $
```

Lessons:

- Indexing must be in the range of existing sequence elements
- Indices outside of this range lead to an **IndexError**

Data Types: list

Type **list**

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[2:4] )
4  print( lst[2:5] )
5  print( lst[2:500] )
```

```
1  $ python myscript.py
```

Data Types: list

Type **list**

myscript.py

```
1  lst = [1, 3, 4, 5]
2
3  print( lst[2:4] )
4  print( lst[2:5] )
5  print( lst[2:500] )
```

```
1  $ python myscript.py
2  [4, 5]
3  [4, 5]
4  [4, 5]
5  $
```

Lessons:

- Indices that are out of range do not lead to an error in slicing
- Simple explanation: a sequence range can also be empty

Data Types: bool

- The boolean type represents either **True** or **False**
- As simple as it seems: it is of great importance (be patient)

Data Types: bool

- The boolean type represents either **True** or **False**
- As simple as it seems: it is of great importance (be patient)

myscript.py

```
1  a = True
2  b = False
3
4  print(a)
5  print(b)
6
7  print( type(a) )
8  print( type(b) )
```

```
1  $ python myscript.py
```

Data Types: bool

- The boolean type represents either **True** or **False**
- As simple as it seems: it is of great importance (be patient)

myscript.py

```
1  a = True
2  b = False
3
4  print(a)
5  print(b)
6
7  print( type(a) )
8  print( type(b) )
```

```
1  $ python myscript.py
2  True
3  False
4  <class 'bool'>
5  <class 'bool'>
6  $
```

Data Types: Conversion

- There are (built-in) functions to convert some data types: **bool()**, **int()**, **float()**, **str()**

myscript.py

```
a = 10
b = float(a)
print( b )
print( type(b) )

c = 10.0
d = int(c)
print( d )
print( type(d) )
```

```
$ python myscript.py
```

Data Types: Conversion

- There are (built-in) functions to convert some data types:
  bool(), int(), float(), str()

myscript.py

```
a = 10
b = float(a)
print( b )
print( type(b) )

c = 10.0
d = int(c)
print( d )
print( type(d) )
```

```
$ python myscript.py
10.0
<class 'float'>
10
<class 'int'>
$
```

Data Types: Conversion

myscript.py

```
a = 10
b = str(a)
print( b )
print( type(b) )

c = "10"
d = int(c)
print( d )
print( type(d) )
```

```
$ python myscript.py
```

Data Types: Conversion

myscript.py

```
a = 10
b = str(a)
print( b )
print( type(b) )

c = "10"
d = int(c)
print( d )
print( type(d) )
```

```
$ python myscript.py
10
<class 'str'>
10
<class 'int'>
$
```

Data Types: Conversion

myscript.py

```
a = 10.1
b = str(a)
print( b )
print( type(b) )

c = "10.1"
d = float(c)
print( d )
print( type(d) )
```

```
$ python myscript.py
```

Data Types: Conversion

myscript.py

```
1  a = 10.1
2  b = str(a)
3  print( b )
4  print( type(b) )
5
6  c = "10.1"
7  d = float(c)
8  print( d )
9  print( type(d) )
```

```
1  $ python myscript.py
2  10.1
3  <class 'str'>
4  10.1
5  <class 'float'>
6  $
```

Data Types: Conversion

myscript.py

```
1  print( int(11.1) )
2  print( int(11.5) )
3  print( int(11.9) )
```

```
1  $ python myscript.py
```

Data Types: Conversion

myscript.py

```
1  print( int(11.1) )
2  print( int(11.5) )
3  print( int(11.9) )
```

```
1  $ python myscript.py
2  11
3  11
4  11
5  $
```

Lesson: conversion from float to int **looses precision**

Data Types: Conversion

myscript.py

```
1  print( int("11.1") )
```

```
1  $ python myscript.py
```

Data Types: Conversion

myscript.py

```
1  print( int("11.1") )
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "myscript.py", line 1, in <module>
4      print( int("11.1") )
5  ValueError: invalid literal for int() with base 10: '11.1'
6  $
```

Data Types: Conversion

myscript.py

```
1  print( int("11.1") )
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "myscript.py", line 1, in <module>
4      print( int("11.1") )
5  ValueError: invalid literal for int() with base 10: '11.1'
6  $
```

myscript.py

```
1  print( float("a") )
```

```
1  $ python myscript.py
```

Data Types: Conversion

myscript.py

```
1  print( int("11.1") )
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "myscript.py", line 1, in <module>
4      print( int("11.1") )
5  ValueError: invalid literal for int() with base 10: '11.1'
6  $
```

myscript.py

```
1  print( float("a") )
```

```
1  $ python myscript.py
2  Traceback (most recent call last):
3    File "myscript.py", line 1, in <module>
4      print( float("a") )
5  ValueError: could not convert string to float: 'a'
6  $
```

Lesson: be careful and ensure that source content can be converted to destination type

Basics: Comparison Operators

- These operators can be used to compare objects (the operands)
- These so-called **expressions evaluate to** (return) bool

Basics: Comparison Operators

- These operators can be used to compare objects (the operands)
- These so-called **expressions evaluate to** (return) bool

myscript.py

```
1  a = 1
2  b = 2
3  print( a == b)
4  print( a != b)
5  print( a <  b)
6  print( a >  b)
7  print( a <= b)
8  print( a >= b)
```

Basics: Comparison Operators

- These operators can be used to compare objects (the operands)
- These so-called **expressions evaluate to** (return) bool

myscript.py

```
1  a = 1
2  b = 2
3  print( a == b)
4  print( a != b)
5  print( a <  b)
6  print( a >  b)
7  print( a <= b)
8  print( a >= b)
```

```
1  $ python myscript.py
2  False
3  True
4  True
5  False
6  True
7  False
8  $
```

Basics: Comparison Operators

myscript.py

```
a = "r"
b = "s"
print( a == b)
print( a != b)
print( a <  b)
print( a >  b)
print( a <= b)
print( a >= b)
```

Basics: Comparison Operators

myscript.py

```
1  a = "r"
2  b = "s"
3  print( a == b)
4  print( a != b)
5  print( a <  b)
6  print( a >  b)
7  print( a <= b)
8  print( a >= b)
```

```
1  $ python myscript.py
2  False
3  True
4  True
5  False
6  True
7  False
8  $
```

Lesson: comparison operators also work on type str

Basics: Comparison Operators

myscript.py

```
1  a = "charly"
2  b = "charlz"
3  print( a ==  b)
4  print( a <  b)
5  print( a >  b)
```

Basics: Comparison Operators

myscript.py

```
1  a = "charly"
2  b = "charlz"
3  print( a ==  b)
4  print( a <  b)
5  print( a >  b)
```

```
1  $ python myscript.py
2  False
3  True
4  False
5  $
```

Lesson: comparison operators on type str evaluate lexicographically

Basics: Boolean Operators

- These operators are also called logical operators: **not**, **and**, **or**
- The above ordering has descending **priority**
- The resulting expressions evaluate to bool

myscript.py

```
1 print(not True)
2 print(not False)
3 print(True and True)
4 print(True and False)
5 print(True or True)
6 print(True or False)
7 print(True or not False)
```

Basics: Boolean Operators

- These operators are also called logical operators: **not**, **and**, **or**
- The above ordering has descending **priority**
- The resulting expressions evaluate to bool

myscript.py

```
1  print(not True)
2  print(not False)
3  print(True and True)
4  print(True and False)
5  print(True or True)
6  print(True or False)
7  print(True or not False)
```

```
1  $ python myscript.py
2  False
3  True
4  True
5  False
6  True
7  True
8  True
9  $
```

Basics: Compound Expressions

- Expressions can be combined by chaining and nesting
- Arbitrary complex compound expressions can be build

myscript.py

```
1  a = 3
2  print(5 + a + 2)
3  print("5" + "3" + "2")
4  print(2 < a <=3)
5  print(True and True and False)
6  print(False or False or True)
```

```
1  $ python myscript.py
```

Basics: Compound Expressions

- Expressions can be combined by chaining and nesting
- Arbitrary complex compound expressions can be build

myscript.py

```
1  a = 3
2  print(5 + a + 2)
3  print("5" + "3" + "2")
4  print(2 < a <=3)
5  print(True and True and False)
6  print(False or False or True)
```

```
1  $ python myscript.py
2  10
3  532
4  True
5  False
6  True
7  $
```

Basics: Compound Expressions

- Of great importance is the **precedence** of operators
- Operators in Python have an assigned precedence
- Precedence defines the order of evaluation in compound expression
- Operators with higher precedence are evaluated first

myscript.py

```
1  print(5 + 3 * 2)
2  print("5" + "3" * 2)
3  print()
```

```
1  $ python myscript.py
```

Basics: Compound Expressions

- Of great importance is the **precedence** of operators
- Operators in Python have an assigned precedence
- Precedence defines the order of evaluation in compound expression
- Operators with higher precedence are evaluated first

myscript.py

```
1  print(5 + 3 * 2)
2  print("5" + "3" * 2)
3  print()
```

```
1  $ python myscript.py
2  11
3  533
4  $
```

Basics: Compound Expressions

- Of great importance is the **precedence** of operators
- Operators in Python have an assigned precedence
- Precedence defines the order of evaluation in compound expression
- Operators with higher precedence are evaluated first

myscript.py

```
1  print(5 + 3 * 2)
2  print("5" + "3" * 2)
3  print()
```

```
1  $ python myscript.py
2  11
3  533
4  $
```

Lessons: '*' has higher precedence than '+'

Basics: Compound Expressions

- A non-exhaustive list of important operators ordered by precedence

Precedence	Operator	Description
lowest	or	boolean or
	and	boolean and
	not	boolean not
	==, !=, <, >, <=, >=	comparison operators
	+, -	addition, subtraction
	*, /, %	multiplication, division, modulo
	+x, -x	positive, negative x
highest	**	exponentiation

Basics: Compound Expressions

- Parentheses allow to change order of evaluation by precedence
- Expressions in parentheses are evaluated first: it's like maths ;)
- Parentheses can also be used to just enhance readability

myscript.py

```
1  print(  5 + 3  * 2 )
2  print( (5 + 3) * 2 )
3  print(  "5" + "3"  * 2 )
4  print( ("5" + "3") * 2 )
5
6  print( 5 * 3 + 3 * 2 )
7  print( (5 * 3) + (3 * 2) )
```

```
1  $ python myscript.py
```

Basics: Compound Expressions

- Parentheses allow to change order of evaluation by precedence
- Expressions in parentheses are evaluated first: it's like maths ;)
- Parentheses can also be used to just enhance readability

myscript.py

```
1  print(  5 + 3  * 2 )
2  print( (5 + 3) * 2 )
3  print(  "5" + "3"  * 2 )
4  print( ("5" + "3") * 2 )
5
6  print( 5 * 3 + 3 * 2 )
7  print( (5 * 3) + (3 * 2) )
```

```
1  $ python myscript.py
2  11
3  16
4  533
5  5353
6  21
7  21
8  $
```

Basics: Comparison Pitfalls

- **Attention**: there's an operator **is**
- It has not the same meaning as the equality operator '=='

myscript.py

```
1  a = [1, 2, 3]
2  b = [1, 2, 3]
3  print(a == b)
4  print(a is b)
```

```
1  $ python myscript.py
```

Basics: Comparison Pitfalls

- **Attention**: there's an operator **is**
- It has not the same meaning as the equality operator '=='

myscript.py

```
1  a = [1, 2, 3]
2  b = [1, 2, 3]
3  print(a == b)
4  print(a is b)
```

```
1  $ python myscript.py
2  True
3  False
4  $
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3  print(a == b)
```

```
1  $ python myscript.py
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3  print(a == b)
```

```
1  $ python myscript.py
2  False
3  $
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3  print(a == b)
4
5  print(f'{a}')
6  print(f'{b}')
```

```
1  $ python myscript.py
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3  print(a == b)
4
5  print(f'{a}')
6  print(f'{b}')
```

```
1  $ python myscript.py
2  False
3  1.4000000000000001
4  1.4
5  $
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!
- **Solution**: test if floats are close enough: difference below cutoff

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3
4  tolerance = 0.000001
5  print( abs(a-b) < tolerance )
6
7  tolerance = 0.0000000000000001
8  print( abs(a-b) < tolerance )
```

```
1  $ python myscript.py
```

Basics: Comparison Pitfalls

- **Attention**: equality testing for type float
- Using the equality operator '==' is no solution!
- **Solution**: test if floats are close enough: difference below cutoff

myscript.py

```
1  a = 1.1 + 0.3
2  b = 1.2 + 0.2
3
4  tolerance = 0.000001
5  print( abs(a-b) < tolerance )
6
7  tolerance = 0.0000000000000001
8  print( abs(a-b) < tolerance )
```

```
1  $ python myscript.py
2  True
3  False
4  $
```

Diving Deeper ...

Data Types

→ https://www.w3schools.com/python/python_for_loops.asp

Comparison

→ https://www.w3schools.com/python/python_functions.asp

Practice Time ... 003

1. p001: Find out how the built-in function **round()** rounds to integer.

2. p002: Create a non-empty list.
   Try to access elements with indices larger than the number of elements.
   Study the error message.

3. p003: Experiment and familiarize with the slicing of lists.

4. p004: Do you remember that type **str** is also a sequence?
   Probably slicing also works here, too? Give it a try!

5. p005: Slicing can also have the following syntax 'sequence[n1:n2:n3]'.
   Try to find out what the third number (n3) can be used for.

6. p003: Lists can be 'nested' to form multidimensional lists (matrices).
   Try to find out how it works and implement a few lines to test it.

7. p006: Another important sequence type is called **dict**. → https://realpython.com/python-dicts/
   → Familiarize yourself with it by implementing a small example.

License and Contributors

Main Author
Philipp Thiel

Additional Contributors
n.a.