Computermethoden: Computer Vision

Part 6: Python Programming

Leonid Kostrykin   Karl Rohr

Biomedical Computer Vision Group (BMCV)

BioQuant, IPMB, Heidelberg University

Contents

Introduction to Python programming:

- Python resources
- Characteristics of Python
- Basic elements: Data types, variables, operators
- Control structures: Selection, iteration
- Re-usable code: Functions, modules
- Data structures: Lists, multi-dimensional arrays
- Understanding errors & common mistakes

Python resources

Official Python 3 tutorial:

- `https://docs.python.org/3/tutorial/`

Python 3 course on codecademy:

- `https://www.codecademy.com/learn/learn-python-3`
- Paid subscription required, but free for 7 days

Further resources:

- `https://jupyter.org` – IDE for data analysis
- `https://www.anaconda.com` – Recommended software for installing/managing Python and additional modules

Python in 2019

- Python was created by **Guido van Rossum** and first released in 1991
- Python 2 was released in 2000 and will be discontinued in 2020
- **Python Software Foundation** launched 2001, is a nonprofit organization devoted to continued development of the Python programming language
- Python 3 was first released in 2008
- **Freely available!**

Python rates as the most wanted programming language

Stack Overflow developer survey 2019: Python more popular than Java

April 10, 2019 Sarah Schlothauer

#python

Stack Overflow's annual developer survey is back with results for 2019. Find out what technology is most loved, most dreaded, and most wanted. This year there are more insights about the global developer profile, demographics, and what challenges get in the way of workflow. With nearly 90,000 responses from around the globe, this is the world's largest developer survey.

© Shutterstock / NicoElNino

https://jaxenter.com/stack-overflow-dev-survey-2019-157815.html

Characteristics of Python

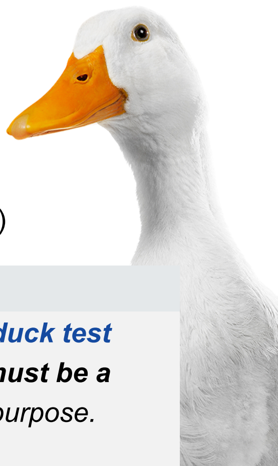
High-level general-purpose programming language

- Interpreted, dynamically typed, garbage-collected
- Very easy to write and understand
  - English keywords
  - Object-oriented and duck-typed
- Well-suited for image and data analysis
- Widely used in scientific computing

Python: Object-oriented and duck-typed

Python is **object-oriented**. What does this mean?

- Python programs consist of **objects** which
  - perform certain tasks
  - interact with each other
- An **object** comprises **attributes** (data) and **methods** (behaviours)



Python is duck-typed

> *Duck typing in computer programming is an application of the **duck test** – **"If it walks like a duck and it quacks like a duck, then it must be a duck"** – to determine if an object can be used for a particular purpose.*

– https://en.wikipedia.org/wiki/Duck_typing

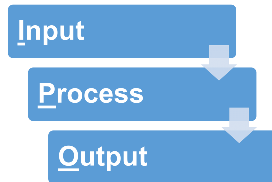
Objects and types

- An object (e.g., "Donald") is instantiated (created) from a **type** (e.g., "Duck")
- An object's **type** defines
  - **which** methods the **object** offers (e.g., walking, quaking, …)
  - **which** attributes the **object** possesses (e.g., age, color, …)
  - **…but not their values!** (e.g., Donald has a *different* age and color than Daisy)

Python Programming

Input-process-output (IPO) model of a program

1. Loads input data (e.g., an image)
2. Processes the loaded data
3. Outputs the processed data (e.g., analysis results)



Cycle of Python programming steps

1. Write Python code
2. Run the written Python code
3. Compare the output results with the expected results
4. If something is wrong: **(this is the usual case!)**
   Critically re-consider your code → Fix errors → Return to step 2

A simple Python program

```
print('Welcome to Python programming!')
```

- `print` is a **built-in function**
- Prints *one line of text* to the **standard output** (e.g., the screen)
- The part between "(" and ")" are the **function parameters**
  - For `print`, this is the **content to be printed**
  - Here a **string** (text, must be put in quotes like "'…'") is printed
- In strings, special characters can be inserted:

`\\`	backslash character	`\t`	tab stop
`\n`	line break	`\'`	single quote

Data types, variables, and operators

Variables, assignments, and expressions

Assignment syntax

```
variable = value
```

Assigns `value` to `variable`, where `value` can be

- An **object** (e.g., number, string, list, duck, …)
- An **expression** which *evaluates* to an object
  (e.g., the expression "`6 + 5`" evaluates to the numeric object `11`)

The `variable` then **represents** the object which it was assigned

Combined expressions: The expression "`3 * (6 + 5)`" evaluates to `33`

Numbers, strings, and variables

```
var1 = 2
var2 = 11
var3 = var1 + var2
print(var3)
var1 = var3
print(var1 * 2 + 0.5)
```

Output

```
13
26.5
```

```
var1 = '42'
var2 = 'What is the answer?'
var3 = var2 + ' ' + var1
print(var3)
```

Reminder: String-type objects *start and end* with single quotes!

Output

```
What is the answer? 42
```

Numbers, strings, and variables

```
var1 = 2
var2 = 11
var3 = var1 + var2
print(var3)
var1 = var3
print(var1 * 2 + 0.5)
```

Output

```
13
26.5
```

```
var1 = ’42’
var2 = ’What is the answer?’
var3 = var2 + ’ ’ + var1
print(var3)
```

Reminder: String-type objects *start and end* with single quotes!

Output

```
What is the answer? 42
```

⇒ The **behaviour** of the “ + ” operator depends on the **type** of the operands!

Built-in types

In Python, **numbers** are represented by one of two types

- An object of type `float` can be **any decimal number** between $\pm 1.8 \cdot 10^{308}$ but **most numbers cannot be represented exactly** (limited machine precision)
- An object of type `int` can be **any integer number** (e.g., `-123`, `0`, `321`, …)

Examples for objects of type `float`

`-123.2, 1.2e3, 0.0, 12.0, 3.14, …`

Examples for objects of type `int`

`-123, 0, 321, …`

Many other types: `str`, `list`, `dict`, `set`, `frozenset`, `complex`, `bool`, …

Operators and types

Different **operations** (behaviours) performed by different **operators** and **types**:

Pythonic expression	op1 and op2 are **numbers** (e.g., `float` or `int`)	op1 is a **list** or **string** and op2 is… an `int`	op1 is a **list** or **string** and op2 is… **same type** as op1
`op1 + op2`	**sum** of op1 and op2	*undefined*	**concatenation** of op1 and op2
`op1 - op2`	**difference** of op1 and op2	*undefined*	
`op1 * op2`	**product** of op1 and op2	op1 **repeated** op2 times	*undefined*
`op1 / op2`	(true) **division** of op1 and op2	*undefined*	
`op1 % op2`	**modulo** of op1 and op2	*undefined*	
`op1 // op2`	**integer division** of op1 and op2	*undefined*	

Note: "`op1 += op2`" is a shorthand for "`op1 = op1 + op2`". The same holds analogously for the other operators (`-=`, `*=`, `/=`, `%=`, `//=`).

True division vs integer division

True division

- “`6 / 3`” yields “`2.0`” (`float`)
- “`7 / 4`” yields “`1.75`” (`float`)
- “`17 / 5`” yields “`3.4`” (`float`)

Integer division: Fractal part truncated

- “`6 // 3`” yields “`2`” (`int`)
- “`7 // 4`” yields “`1`” (`int`)
- “`17 // 5`” yields “`3`” (`int`)

Modulo operator: Remainder after integer division (fractal part nominator)

- “`6 % 3`” yields “`0`” (`int`, fractal part: $0/4 = 0$)
- “`7 % 4`” yields “`3`” (`int`, fractal part: $3/4 = 0.75$)
- “`17 % 5`” yields “`2`” (`int`, fractal part: $2/5 = 0.4$)
- **Question:** *Is “`op1 // op2`” a shorthand for “`(op1 - op1 % op2) / op2`”?*
- **Application example:** *Determine whether one number is a multiple of the other*

Other mathematical operators

Remember: **The operators** `+`, `-`, `*`, `%`, `//` yield an object of type `int` if op1 and op2 both are of type `int`, and `float` otherwise

Examples

- "`6 + 2`" yields "`8`" (`int`)
- "`6 + 2.0`" yields "`8.0`" (`float`)

Pythonic expression	Type of the result	
	op1 **or** op2 is `float`	op1 **and** op2 are `int`
`op1 + op2`	`float`	`int`
`op1 - op2`	`float`	`int`
`op1 * op2`	`float`	`int`
`op1 / op2`	`float`	`float`
`op1 % op2`	`float`	`int`
`op1 // op2`	`float`	`int`

Why is this important?

Example: Indexing – *see later, be patient!*

String formatting

```
day   = 31
month = ’January’
year  = 2019
```

Task: Print the above variables in a canonical date format (“`31. January 2019`”)

Is this a correct solution?

```
print(day + ’. ’ + month + ’ ’ + year)
```

String formatting

```
day   = 31
month = ’January’
year  = 2019
```

Task: Print the above variables in a canonical date format (“`31. January 2019`”)

Is this a correct solution?

```
print(day + ’. ’ + month + ’ ’ + year)
```

It is wrong! The “ + ”-operators are used for operands of *different* types (**numbers** and **strings**). This case is *undefined!*

String formatting

```
day   = 31
month = ’January’
year  = 2019
```

Task: Print the above variables in a canonical date format (“`31. January 2019`”)

Correct solutions:

```
print(str(day) + ’. ’ + str(month) + ’ ’ + str(year))
```

```
print(f’{day}. {month} {year}’)
```

- “**f**”-prefix creates a **formatted string**
- Substitutes “`{day}`” by the value of the variable “day”, …

```
print(’%d. %s %d’ % (day, month, year))
```

C-style string formatting (old-fashioned)

Selections and iterations

Selections

Structures of the `if`-statement permitted in Python

```
if condition:
    instruction1
    instruction2
```

```
if condition1:
    instruction1
    instruction2
else:
    instruction3
    instruction4
```

```
if condition1:
    instruction1
    instruction2
elif condition2:
    instruction3
    instruction4
else:
    instruction5
    instruction6
```

```
if condition1: instruction1
elif condition2: instruction2
else: instruction3
```

Conditions

Conditions can be implemented as

- An object of type `bool` (either of the two built-in objects “`True`” or “`False`”)
- A variable which *represents* an object of type `bool` – **Example:**

```
var1 = True
if var1: print('var1 is True')
```

- An expression which *evaluates* to an object of type `bool` (e.g., uses *equality or relational operators*) – **Example:**

```
var1 = True
if not var1: print('var1 is False')
```

Equality or relational operators

Expressions which evaluate to an object of type `bool`:

Pythonic expression	Evaluates to `True` if… (and `False` otherwise)
Equality operators	
`op1 == op2`	…op1 is **equal** to op2
`op1 != op2`	…op1 is **not equal** to op2 (equivalent to "`not op1 == op2`")
Relational operators	
`op1 < op2`	…op1 is **less** than op2
`op1 > op2`	…op1 is **greater** than op2
`op1 <= op2`	…op1 is **less equal than or equal** to op2
`op1 >= op2`	…op1 is **greater than or equal** to op2
`op1 in op2`	…op2 **contains** op1 (only defined if op2 is a string, list, …)

Example for `if`-conditions

```
student_grade = 25
s1 = 'Passed!'
s2 = 'Failed!'

student_grade *= 2

if student_grade >= 50: res = s1
else: res = s2

res = f"Grade={student_grade} ==> {res}"
print(res)
```

Example for if-conditions

```
student_grade = 25
s1 = ’Passed!’
s2 = ’Failed!’

student_grade *= 2

if student_grade >= 50: res = s1
else: res = s2

res = f”Grade={student_grade} ==> {res}”
print(res)
```

Output

```
Grade=50 ==> Passed!
```

Iterations

Structures of the while-loop in Python

```
while condition:
    instruction1
    instruction2
    ...
```

Example

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Output

```
0
1
2
...
9
```

The for-loop and iterables

The while-loop from the previous slide can be formulated as follows

Structures of the for-loop in Python

```
for item in iterable:
    instruction1
    instruction2
    ...
```

Example

```
for i in range(0, 10):
    print(i)
```

Indentation: 4× whitespace! (or ⇆)

An **iterable** can be any **object which contains items**, for example:

- A **range** of `int` objects, `range(a, b)` corresponds to the *half-closed* interval $\mathbb{Z} \cap [a, b)$. **Note: a and b must be of type `int`!**
- A **list** (sequence of arbitrary objects), a **string** (sequence of characters), …
- More examples will be learned during the *practical lab sessions!*

More loop examples

Example 1

```
product = 2
while product <= 10_000:
    product *= 2
print(product)
```

Hint: “10_000” is a more easily human-readable notation for 10000

Example 2

```
result = ’’
for character in ’Hello World’:
    result = character + result
print(result)
```

More loop examples

Example 1

```
product = 2
while product <= 10_000:
    product *= 2
print(product)
```

Hint: "10_000" is a more easily human-readable notation for 10000

Output: 16384

Example 2

```
result = ''
for character in 'Hello World':
    result = character + result
print(result)
```

More loop examples

Example 1

```
product = 2
while product <= 10_000:
    product *= 2
print(product)
```

Hint: “10_000” is a more easily human-readable notation for 10000

Output: 16384

Example 2

```
result = ’’
for character in ’Hello World’:
    result = character + result
print(result)
```

Output: dlroW olleH

Functions and modules

Re-usable code

Python supports **code re-usability** at different levels:

1. **Classes** (e.g., specify the *blueprint* of a duck, i.e., its *attributes* and how it *behaves*, then synthesize as many ducks as you want)
2. **Functions** (e.g., specify a receipt for how to roast ~~a duck~~ tofu *once*, then roast many ~~ducks~~ tofus on an assembly line)
3. **Modules:** Collections of classes + functions for specific tasks (e.g., load and use the *duck processing toolkit* someone else wrote, or write your own one)

Re-usable code

Python supports **code re-usability** at different levels:

1. **Classes** (e.g., specify the *blueprint* of a duck, i.e., its *attributes* and how it *behaves*, then synthesize as many ducks as you want)
2. **Functions** (e.g., specify a receipt for how to roast ~~a duck~~ tofu *once*, then roast many ~~ducks~~ tofus on an assembly line)
3. **Modules:** Collections of classes + functions for specific tasks (e.g., load and use the *duck processing toolkit* someone else wrote, or write your own one)

Functions

Built-in functions

- `abs(number)` – computes $|x|$ where x is a number
- `int(number)` – truncates the fractal part of `number` (yields an `int` object)
- `len(iterable)` – counts items in `iterable` (e.g., characters in a string)
- `pow(a, b)` – computes a^b for any numbers `a` and `b`
- `print(value)` – prints `value` to the standard output
- `range(a, b)` – yields an iterable for `int` objects in $[a, b) \cap \mathbb{Z}$
- `round(number)` – rounds `number` to the closest integer value (yields `int`)
- `str(value)` – converts `value` (e.g., `float`, `int`) to a string
- `sum(iterable)` – yields sum of items in `iterable` (e.g., numbers in a list)
- *and many others…*

Writing functions

Lets define a **re-usable function** “`reverse`” which **reverses a string**

Syntax for function definitions

```
def function(parameters):
    instruction1
    instruction2
    ...
```

Example

```
def reverse(string):
    result = ''
    for character in string:
        result = character + result
    return result
```

Indentation: 4× whitespace! (or **)**

- The `return` instruction determines the object, which the function evaluates to
- The `return` instruction terminates the execution of the function
- The function evaluates to the object `None` if no `return` instruction is encountered

Examples for self-written functions

```
def reverse(string):
    result = ’’
    for character in string:
        result = character + result
    return result

reverse(’Hello World’)
reverse(reverse(’Hello World’))
```

Examples for self-written functions

```
def reverse(string):
    result = ’’
    for character in string:
        result = character + result
    return result

reverse(’Hello World’)
reverse(reverse(’Hello World’))
```

Output

```
dlroW olleH
```

Examples for self-written functions

```
def reverse(string):
    result = ’’
    for character in string:
        result = character + result
    return result

reverse(’Hello World’)
reverse(reverse(’Hello World’))
```

Output

```
dlroW olleH

Hello World
```

Modules

An ecosystem that can hardly be overlooked

Popular low-level Python modules

- `numpy` – numerical Python extensions (linear algebra, multi-dimensional arrays, …)
- `scipy` – scientific computing (e.g., optimization, interpolation, image processing, …)

Popular high-level Python modules

- `sklearn` – machine learning
- `skimage` – image processing (stand-alone + integrates into `sklearn`)

Loading and using modules

Fundamentals of working with modules

Purpose	Syntax	Example
Loading a module	`import module`	`import math`
Using an object from a loaded module	`module.object`	`print(math.pi)`
Using a function from a loaded module	`module.function(…)`	`print(math.log10(10))`

Hierarchical organization: Some modules contain sub-modules – Example:

```
import skimage.io
img = skimage.io.imread('filepath.tiff')
```

Reads the image from the file "`filepath.tiff`" as an object represented by `img`

Lists and multi-dimensional arrays

Lists

- **So far:** A **single variable** represents a **single object**

`variable` ←(assign)— `object`, e.g.: `var1 = 12`

- **A list** is an **object** which contains references to **multiple objects**

`list (object)` ←(append)— `object`, e.g.:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

Lists

- **So far:** A **single variable** represents a **single object**

variable ←(assign)— object , e.g.: `var1 = 12`

- **A list** is an **object** which contains references to **multiple objects**

list (object) ←(append)— object , e.g.:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

Lists

- **So far:** A **single variable** represents a **single object**

`variable` ← (assign) `object` , e.g.: `var1 = 12`

- **A list** is an **object** which contains references to **multiple objects**

`list (object)` ← (append) `object` , e.g.:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

- **Removing an object** from a list deletes the reference (the object still exists)

Example:

```
var1.remove(14)
print(var1)
```

produces the output: `[12, 15]`

list indexing and iterating

A list is a continuous sequence of object references (*"flat map of positions to objects"*)

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
var1.append(-3)
```

Position	Object
0	12
1	14
2	15
3	-3

Accessing data in a list

- **`len(var1)`**: Yields number of items in `var1`
- **`var1[ i ]`**: Represents the `i`-th item (entry) of the list ("`i`" is called *position* or *index*)

  Notes:
  - `i` must suffice $-$`len(var1)` $\leq$ `i` $<$ `len(var1)`
  - `i = -1` refers to the last item
  - `i = -2` to the one before the last, …
  - `i` must be an `int`

Iteration by index

```
for i in range(0, len(var1)):
    print(var1[i])
```

Lists are iterable

```
for obj in var1:
    print(obj)
```

Output

```
12
14
15
-3
```

list indexing and iterating

A list is a continuous sequence of object references (*"flat map of positions to objects"*)

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
var1.append(-3)
```

Position	Object
0	12
1	14
2	15
3	-3

Accessing data in a list

- `len(var1)`: Yields number of items in `var1`
- `var1[ i ]`: Represents the `i`-th item (entry) of the list ("`i`" is called *position* or *index*)

Notes:
- `i` must suffice $-$`len(var1)` $\leq$ `i` $<$ `len(var1)`
- `i = -1` refers to the last item
- `i = -2` to the one before the last, …
- `i` must be an `int`

Iteration by index

```
for i in range(0, len(var1)):
    print(var1[i])
```

Lists are iterable

```
for obj in var1:
    print(obj)
```

Output

```
12
14
15
-3
```

list indexing and iterating

A list is a continuous sequence of object references (*"flat map of positions to objects"*)

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
var1.append(-3)
```

Position	Object
0	12
1	14
2	15
3	-3

Accessing data in a list

- **`len(var1)`**: Yields number of items in `var1`
- **`var1[ i ]`**: Represents the `i`-th item (entry) of the list ("`i`" is called *position* or *index*)

  Notes:
  - `i` must suffice $-$`len(var1)` ≤ `i` < `len(var1)`
  - `i = -1` refers to the last item
  - `i = -2` to the one before the last, …
  - `i` must be an `int`

Iteration by index

```
for i in range(0, len(var1)):
    print(var1[i])
```

Lists are iterable

```
for obj in var1:
    print(obj)
```

Output

```
12
14
15
-3
```

list indexing and iterating

A list is a continuous sequence of object references (*"flat map of positions to objects"*)

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
var1.append(-3)
```

Position	Object
0	12
1	14
2	15
3	-3

Accessing data in a list

- **`len(var1)`**: Yields number of items in `var1`
- **`var1[i]`**: Represents the `i`-th item (entry) of the list ("`i`" is called *position* or *index*)

  Notes:
  - `i` must suffice $-$`len(var1)` ≤ `i` < `len(var1)`
  - `i = -1` refers to the last item
  - `i = -2` to the one before the last, …
  - `i` must be an `int`

Iteration by index

```
for i in range(0, len(var1)):
    print(var1[i])
```

Lists are iterable

```
for obj in var1:
    print(obj)
```

Output

```
12
14
15
-3
```

list indexing example

Note:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

is equivalent to

```
var1 = [12, 14, 15]
```

list indexing example

Note:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

is equivalent to

```
var1 = [12, 14, 15]
```

Example

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) / 2])
```

list indexing example

Note:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

is equivalent to

```
var1 = [12, 14, 15]
```

Example (error)

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) / 2])
```

list indexing example

Note:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

is equivalent to

```
var1 = [12, 14, 15]
```

Example (error)

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) / 2])
```

"`len(data) / 2`" **evaluates to "`3.0`" (`float`)** but "`data[ … ]`" **expects an** `int` object! – *cf. Slides 13 and 30*

Example (corrected)

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) // 2])
```

list indexing example

Note:

```
var1 = list()
var1.append(12)
var1.append(14)
var1.append(15)
```

is equivalent to

```
var1 = [12, 14, 15]
```

Example (error)

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) / 2])
```

"`len(data) / 2`" **evaluates to "`3.0`" (`float`)** but "`data[ … ]`" **expects an** `int` object! – *cf. Slides 13 and 30*

Example (corrected)

```
data = [3, 1, 6, 11, 5, 2]
print(data[len(data) // 2])
```

Output

```
11
```

Assignment using `list` indexing

Indexing can also be used for manipulation of lists

Example 1

```
data = [3, 1, 6, 11, 5, 2]
data[2] = -5
print(data)
```

Assignment using `list` indexing

Indexing can also be used for manipulation of lists

Example 1

```
data = [3, 1, 6, 11, 5, 2]
data[2] = -5
print(data)
```

Output

```
[3, 1, -5, 11, 5, 2]
```

Assignment using `list` indexing

Indexing can also be used for manipulation of lists

Example 1

```
data = [3, 1, 6, 11, 5, 2]
data[2] = -5
print(data)
```

Output

```
[3, 1, -5, 11, 5, 2]
```

Example 2

```
data = [3, 1, 6, 11, 5, 2]
data[2] = data[0]
print(data)
```

Assignment using `list` indexing

Indexing can also be used for manipulation of lists

Example 1

```
data = [3, 1, 6, 11, 5, 2]
data[2] = -5
print(data)
```

Output

```
[3, 1, -5, 11, 5, 2]
```

Example 2

```
data = [3, 1, 6, 11, 5, 2]
data[2] = data[0]
print(data)
```

Output

```
[3, 1, 3, 11, 5, 2]
```

list slicing

```
data = [1, 6, 3, 1, 1, 2, 1]
```

- **Indexing:** Accesses an item (*element*) of a `list` (e.g., "`data[2]`" yields `3`)
- **Slicing:** Retrieves a `list` (*subset*) of another `list`

Syntax	Example	Slice
`data[start:end]`	`data[1:4]`	`[6, 3, 1]`
`data[start:]`	`data[4:]`	`[1, 2, 1]`
`data[:end]`	`data[:3]`	`[1, 6, 3]`
`data[start:end:step]`	`data[2:7:2]`	`[3, 1, 1]`
`data[::step]`	`data[::3]`	`[1, 1, 1]`

- Slicing can also be used for **manipulation** of lists – **Example:**

```
data[2:5] = [-1, -2, -1]
print(data)
```

Output: `[1, 6, -1, -2, -1, 2, 1]`

Arrays

- An **array** is a **list of fixed length and type** (fundamental programming entity in lower-level programming languages, e.g., C, C++, Java, …)
- A **multi-dimensional array** is an array with multiple axes
  - **Application examples:** Vector, matrix, 2-D or 3-D image, …
  - **In Python:** Represented by objects of the type `numpy.ndarray` (`numpy` module)

Working with arrays

- An **array** is an object of the type `numpy.ndarray`
- **Load the numpy module:**

```
import numpy
```

- **Create a new `float`-type array** filled with zeros:

```
array = numpy.zeros(shape)
```

or

```
array = numpy.zeros(shape, float)
```

  `shape` is a list which specifies the **array size** (e.g., "`[ 32, 64 ]`" corresponds to a 2-D array with 32 rows and 64 columns)

- `array.ndim` evaluates to the **number of the array dimensions**
- `array` is called **flat** if it is single-dimensional (`array.ndim == 1`)

Arrays and lists

Arrays can also be created from lists

Example 1: List of values

```
array1 = numpy.asarray([1, 2, 3])
print(array1)
print(f'Dimensions: {array1.ndim}')
```

Output

```
[1 2 3]
Dimensions: 1
```

Example 2: List of rows

```
array2 = numpy.asarray([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array2)
print(f'Dimensions: {array2.ndim}')
```

Output

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Dimensions: 2
```

Array indexing and slicing

Arrays can be sliced and indexed analogously to lists

Example 1 – Indexing

```
array1 = numpy.asarray([[1, 2], [3, 4]])
print(array1[0, 0])
```

Output

```
1
```

Example 2 – Slicing

```
array1 = numpy.asarray([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array1[:2, 1:])
```

Output

```
[[2 3]
 [5 6]]
```

Example 3 – Slicing

```
array1 = numpy.zeros([3, 3])
array2 = numpy.asarray([[1, 2], [3, 4]])
array1[1:, :2] = array2
print(array1)
```

Output

```
[[0. 0. 0.]
 [1. 2. 0.]
 [3. 4. 0.]]
```

Understanding errors
&
Common mistakes

Tracebacks

- Python provides a so-called *traceback* when an error occurs
- **Reading a traceback** leads to the error you have made (fundamental programming skill)

Example

```
def get_item(data, i):
    return data[i]

print(get_item([1, 6, 3], 3))
```

Tracebacks

- Python provides a so-called *traceback* when an error occurs
- **Reading a traceback** leads to the error you have made (fundamental programming skill)

Example

```
def get_item(data, i):
    return data[i]

print(get_item([1, 6, 3], 3))
```

Output

```
---------------------------------------------------------------
IndexError                          Traceback (most recent call last)
<ipython-input-20-bf91d868e666> in <module>
      2     return data[i]
      3
----> 4 print(get_item([1, 6, 3], 3))

<ipython-input-20-bf91d868e666> in get_item(data, i)
      1 def get_item(data, i):
----> 2     return data[i]
      3
      4 print(get_item([1, 6, 3], 3))

IndexError: list index out of range
```

Tracebacks

- Python provides a so-called *traceback* when an error occurs
- **Reading a traceback** leads to the error you have made (fundamental programming skill)

Example

```
def get_item(data, i):
    return data[i]

print(get_item([1, 6, 3], 3))
```

Output

```
---------------------------------------------------------------------------
IndexError                            Traceback (most recent call last)
<ipython-input-20-bf91d868e666> in <module>
      2     return data[i]
      3
----> 4 print(get_item([1, 6, 3], 3))

<ipython-input-20-bf91d868e666> in get_item(data, i)
      1 def get_item(data, i):
----> 2     return data[i]
      3
      4 print(get_item([1, 6, 3], 3))

IndexError: list index out of range
```

Common mistake

`IndexError`: Accessing list/array items out of range

Common mistakes

Example

```
def reverse(string):
    result = ’’
    for character in string:
    result = character + result
    return result

reverse(’Hello World’)
```

Common mistakes

Example

```
def reverse(string):
    result = ’’
    for character in string:
    result = character + result
    return result

reverse(’Hello World’)
```

Output

```
  File ”<ipython-input-21-e6c02796aba8>”, line 4
    result = character + result
         ^
IndentationError: expected an indented block
```

Common mistakes

Example

```
def reverse(string):
    result = ''
    for character in string:
    result = character + result
    return result

reverse('Hello World')
```

Output

```
  File "<ipython-input-21-e6c02796aba8>", line 4
    result = character + result
         ^
IndentationError: expected an indented block
```

Common mistake

`IndentationError`: Wrong indentation

Common mistakes

Example

```
data = [1, 6, 3, 1, 1, 2, 1]
for i in range(0, len(data)):
    print(fancy_function(data, i))

def fancy_function(data, position):
    return sum(data[:1 + position])
```

Common mistakes

Example

```
data = [1, 6, 3, 1, 1, 2, 1]
for i in range(0, len(data)):
    print(fancy_function(data, i))

def fancy_function(data, position):
    return sum(data[:1 + position])
```

Output

```
---------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-fcf3bd0f1bea> in <module>
      1 data = [1, 6, 3, 1, 1, 2, 1]
      2 for i in range(0, len(data)):
----> 3     print(fancy_function(data, i))
      4
      5 def fancy_function(data, position):

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

Common mistakes

Example

```
data = [1, 6, 3, 1, 1, 2, 1]
for i in range(0, len(data)):
    print(fancy_function(data, i))

def fancy_function(data, position):
    return sum(data[:1 + position])
```

Output

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-fcf3bd0f1bea> in <module>
      1 data = [1, 6, 3, 1, 1, 2, 1]
      2 for i in range(0, len(data)):
----> 3     print(fancy_function(data, i))
      4
      5 def fancy_function(data, position):

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

Common mistake

NameError: Function used before definition

Common mistakes

Example

```
for character in ’Hello World’:
    result = character + result
print(result)
```

Common mistakes

Example

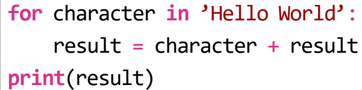
```
for character in ’Hello World’:
    result = character + result
print(result)
```

Output

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-bc66d091382b> in <module>
      1 for character in ’Hello World’:
----> 2     result = character + result
      3 print(result)

NameError: name ’result’ is not defined
```

Common mistakes

Example



Output

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-bc66d091382b> in <module>
      1 for character in ’Hello World’:
----> 2     result = character + result
      3 print(result)

NameError: name ’result’ is not defined
```

Common mistake

`NameError:` Variable usage before assignment

Common mistakes

Example 1

```
data = [3, 1, 6, 11, 5, 2, 1]
for i in range(0, len(data) / 2):
    print(data[i])
```

Common mistakes

Example 1

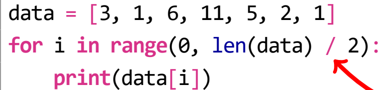
```
data = [3, 1, 6, 11, 5, 2, 1]
for i in range(0, len(data) / 2):
    print(data[i])
```

Output

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-af826dd5c45c> in <module>
      1 data = [3, 1, 6, 11, 5, 2, 1]
----> 2 for i in range(0, len(data) / 2):
      3     print(data[i])
      4

TypeError: ’float’ object cannot be interpreted as an integer
```

Common mistakes

Example 1



```
data = [3, 1, 6, 11, 5, 2, 1]
for i in range(0, len(data) / 2):
    print(data[i])
```

Output

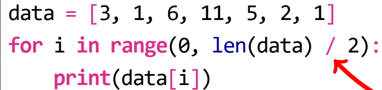
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-af826dd5c45c> in <module>
      1 data = [3, 1, 6, 11, 5, 2, 1]
----> 2 for i in range(0, len(data) / 2):
      3     print(data[i])
      4

TypeError: 'float' object cannot be interpreted as an integer
```

Common mistake

`TypeError:` The object used for `range`, indexing, or slicing is not of type `int`

Common mistakes

Example 1

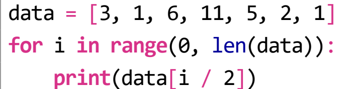


```
data = [3, 1, 6, 11, 5, 2, 1]
for i in range(0, len(data) / 2):
    print(data[i])
```

Output

```
---------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-af826dd5c45c> in <module>
      1 data = [3, 1, 6, 11, 5, 2, 1]
----> 2 for i in range(0, len(data) / 2):
      3     print(data[i])
      4

TypeError: 'float' object cannot be interpreted as an integer
```

Example 2



```
data = [3, 1, 6, 11, 5, 2, 1]
for i in range(0, len(data)):
    print(data[i / 2])
```

Common mistake

`TypeError:` The object used for `range`, indexing, or slicing is not of type `int`

Questions?