JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Introduction to Python for Biologists

Katerina Taškova[1] Jean-Fred Fontaine[1,2]

[1]Faculty of Biology, Johannes Gutenberg-Universität Mainz, Mainz, Germany

[2]Genomics and Computational Biology, Kernel Press, Mainz, Germany

`https://cbdm.uni-mainz.de/mb17`

March 21, 2017

Table of Contents

What is Python?

- Python is a general-purpose programming language
  - created by Guido van Rossum (1991)
  - high-level (abstraction from the details of the computer)
  - interpreted (needs an interpreter software)
- Python design philosophy
  - code readability
  - syntax brevity
- Python is widely used for Biology
  - rich built-in features
  - powerful scientific extensions
  - plotting capabilities

Structured programming I

- **Instructions** are executed sequentially, one per line
- **Conditional statements** allow selective execution of code blocks
- **Loops** allow repeated execution of code blocks
- **Functions** allow on-demand execution of code blocks

Structured programming II

```
instruction 1      # 1st instruction (hashtag # starts comments)
                   # blank line
repeat 20 times    # 2nd instruction (loop starts a block)
    instruction a  # block defined by indentation (spaces or tabs)
    instruction b  # 2nd instruction in block
                   # blank line
if n>10            # 3rd instruction (Conditional statement)
    instruction a  # 1st instruction in block
    instruction b  # 2nd instruction in block
                   # blank line
                   # blank line
# backslashs join lines
instruction 3 \    # 3rd instruction, part 1
instruction 3      # 3rd instruction, part 2
                   # blank line
# Expressions in (), {}, or [] can span multiple lines
instruction 4 (1, 2, 3  # 4th instruction, part 1
               4, 5, 6) # 4th instruction, part 2
```

Namespace

- **Variables** are names associated with data
  - e.g. **a=2** assigns value 2 to variable a
- **Functions** are names associated to specific code blocks
  - built-in functions are available (see list on slide 100)
  - e.g. **print(a)** will display '2' on the screen
- The user **namespace** is the set of names available to the user
  - users can define new names of variables and functions in their namespace
  - imported **modules** can add names of variables and functions in the user namespace

Object-oriented programming

- Data is organized in **classes** and **objects**
  - a **class** is a template defining what objects can store and do
  - an object is an **instance** of a class
  - objects have **attributes** to store data and **methods** to do actions
  - object namespaces are different from user namespace
- Example class "Human" is defined as:
  - has a name (an attribute "name")
  - has an age (an attribute "age")
  - can introduce itself (a method "who")
  - example with 1 existing Human object P1:

```
1 P1.name = "Mary" # assigns value to attribute name
2 P1.age  = 26     # assigns value to attribute age
3 P1.who()         # displays "My name is Mary I am 26!"
4 who()            # error! not in the user namespace
```

Modules

- **Modules** can add functionalities to Python
  - e.g. classes and functions
- Example of available modules:
  - NumPy for scientific computing
  - Matplotlib for plotting
  - BioPython for Biology
- Modules have to be **imported** into the code

```
1 # import datetime module in its own namespace
2 import datetime
3 datetime.date.today() # 2017-03-16
4 today()              # error!
5
6 # import functions log2 and log10 from module math
7 # in current namespace
8 from math import log2, log10
9 log10(1) # equal 0
```

Running code I

- From a terminal by using the interactive Python shell

```
$ python3                    # opens Python shell
a=2                          # assigns 2 to a
b=3                          # assigns 3 to b
exit()                       # closes Python shell
```

- From a terminal by running a script file
  - e.g. let say **myscript.py** is a script file (simple text file)
  - and it contains: **print("hello world!")**

```
$ python3 myscript.py        # runs python3 and the script
hello world!                 # result of the script on the terminal
```

Running code II

- From Jupyter Notebook
  - web-based graphical interface
  - manage **cells** of code or text
  - see execution results on the same notebook
  - save/open notebooks

Documentation and messages I

Documentation and help:

- https://docs.python.org/3
- use the built-in **help()** function
  - e.g. **help(print)** to display help for function **print()**
- see help menu or Google it

Examples of error messages

```
1 # Forgetting quotes
2 print(Hello world)
3 # File "<stdin>", line 2
4 #   print(Hello world)
5 #                     ^
6 # SyntaxError: invalid syntax
```

Documentation and messages II

```
# Spelling mistakes
prin("Hello world")
# Traceback (most recent call last):
#   File "<stdin>", line 2, in <module>
# NameError: name 'prin' is not defined
```

```
# Wrong line break within a string
print("Hello
World")
# File "<stdin>", line 2
#   print("Hello
#                ^
# SyntaxError: EOL while scanning string literal
```

Numeric and strings literals I

```
1  # Numeric literals
2  12
3  -123
4  1.6E3   # means 1600
5
6  # Strings literals
7  'A string'                  # A string
8  'A "string"'                # A "string"
9  "A 'string'"                # A 'string'
10 '''Three single quotes'''   # Three single quotes
11 """Three double quotes"""   # Three double quotes
12 'A \'string\''              # A 'string' (backslash escape sequence)
13 r'A \'string\''             # A \'string\' (raw string)
```

Python stores literals in objects of corresponding classes (class int for integers, float for floatting point, and str for strings)

Numeric and strings literals II

Printing numeric and strings literals

```
1  print(12)  # 12
2  print(1+2) # 3
3
4  print('Hello World') # Hello World
5
6  print('Hello World', 1+2)              # Hello World 3
7  print('Hello World', 1+2, sep='-')     # Hello World-3
8  print('Hello World', 1+2, sep='\t')    # Hello World  3
9                                         # (\t: tab, \n: newline)
10
11 print('AB', end='') # AB (avoid newline at the end)
12 print('CD')         # ABCD
13
14 print('Max is', 12, 'and Min is', 3) # Max is 12 and Min is 3
```

Variables I

Variables are names used to access objects

- first letter is a character (not a digit)
- no space characters allowed
- case-sensitive (variable name var is not Var)
- prefer alphanumeric characters (e.g. abc123)
  - avoid accents, non-alphanumeric, non English
  - underscores may be used (e.g. abc_123)

The following keywords can not be used as variable names

- and, assert, break, class, continue
- def, del, elif, else, except, exec, finally, for, from
- global, if, import, in, is, lambda, not, or, pass
- print, raise, return, try, while, yield

Variables II

```
1  # Numeric types
2  a=2       # a is assigned an int object of value 2
3  print(a) # prints the object assigned to a (2)
4  b=a       # b is assigned the same object as a (2)
5  print(b) # 2
6  a=5       # a is assigned a new object of value 5
7  print(a) # 5
8  print(b) # 2 (b is still assigned to object of value 2)
9
10 # Strings
11 c1='a'
12 print(c1) # 'a'
13 myName125 = 'abc'
```

Numeric types I

```
type(7)        # <class 'int'>   (integer number)
type(8.25)     # <class 'float'> (floating point)
type(4.52e-3) # <class 'float'> (floating point)

# Operators (special built-in functions)
1 + 3  # 4    (addition)
4 - 1  # 3    (substraction)
3 * 2  # 6    (multiplication)
9 / 2  # 4.5  (division)
9 // 2 # 4    (integer division)
9 % 2  # 1    (integer division remainder)
2**3   # 8    (exponent)

# Lowest to highest operators precedence (equal if on same line)
+,-            # Addition, Subtraction
*, /, //, %    # Multiplication, Divisions, Remainder
+x, -x         # Positive, Negative
**             # Exponentiation
```

Numeric types II

```
# Built-in functions
abs(-2.58) # 2.58 (absolute value of x)
round(2.5) # 2    (round to closest integer)

# With variables
a = 1          # 1
b = 1 + 1      # 2
c = a + b      # 3
d = a+c*b      # 7 (precedence of * over +)
d = (a+c)*b # 8 (use parentheses to break precedence)

# Short notations (valid for +, -, *, /, ...)
a += 1 # a = a + 1
a *= 5 # a = a * 5

# Special float values
float('NaN')  # nan  (Not a Number)
float('Inf')  # inf: Infinite positive; -inf: Infinite negative
```

Sequence types

Text sequence type:

- **Strings**: immutable sequences of characters

Basic sequence types:

- **Lists**: mutable sequences
- **Tuples**: immutable sequences
- **Ranges**: immutable sequence of numbers

Sequence operations:

- All sequence types support common sequence operations (slide 98)
- Mutable sequence types support specific operations (slide 99)

Strings I

```
1  # Quotes
2  'A string'                  # A string
3  'A "string"'                # A "string"
4  "A 'string'"                # A 'string'
5  '''Three single quotes'''   # Three single quotes
6  """Three double quotes"""   # Three double quotes
7
8  # Escape sequences (see annexes)
9  "A single quote  '" # A single quote '
10 'A single quote \'' # A single quote '
11 "A tabulation   \t"
12 "A newline      \n"
```

- See other escape sequences in slide 97
- Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal

Strings II

```
# Operators
'pipe' + 'tte' # ='pipette' (concatenation)
'A'*7          # ='AAAAAAA' (replication)
'A'*3 + 'C'*2  # ='AAACC'
'A' + str(2.0) # ='A2.0'    (convert number then concatenate)

# Built-in functions
len('A string of characters') # 22 (length in characters)
type('a') # <class 'str'> (string)

# Slices [start:end:step] (0 is index of first character)
"ABCDEFG"[2:5]   # 'CDE'   (F at index 5 excluded)
"ABCDEFG"[:5]    # 'ABCDE' (from begining)
"ABCDEFG"[5:]    # 'FG'    (to the end)
"ABCDEFG"[-2:]   # 'FG'    (-2 from the end: to the end)
"ABCDEFG"[0:5:2] # 'ACE'   (every second letter with step=2)
```

Strings methods I

Strings are immutable: new objects are created for changes

```
seq = "ACGtCCAgTnAGaaGT"

# Case
seq.capitalize() # 'Acgtccagtnagaagt'
seq.casefold()   # 'acgtccagtnagaagt' (eszett => "ss")
seq.lower()      # 'acgtccagtnagaagt' (eszett => eszett)
seq.swapcase()   # 'acgTccaGtNagAAgt'
seq.upper()      # 'ACGTCCAGTNAGAAGT'

# Search and replace
seq.count('a')             # 2 (case sensitive)
seq.count('G',0, 4)        # 1 (slice start and end indexes)
seq.endswith('GT')         # True
seq.endswith('G',0, 4)     # False (slice start and end indexes)
seq.find('GtC')            # 2 (1st hit index, -1 otherwise)
seq.replace("aa", "tt")    # 'ACGtCCAgTnAGttGT' (case sensitive)
seq.replace("A", "x", 2) # 'xCGtCCxgTnAGaaGT' (2 first hits only)
```

Strings methods II

```
1  seq = "ACGtCCAgTnAGaaGT"
2
3  # Is functions
4  seq.isalnum()    # True  (Are all characters alphanumeric?)
5  seq.isalpha()    # True  (Are all characters alphabetic?)
6  seq.islower()    # False (Are all characters lowercase?)
7  seq.isnumeric()  # False (Are all numeric characters?)
8  seq.isspace()    # False (Are all whitespace characters?)
9  seq.isupper()    # False (Are all characters uppercase?)
10
11 # Join and split
12 "-".join(["A","B"]) # 'A-B'
13 "-".join(seq)       # 'A-C-G-t-C-C-A-g-T-n-A-G-a-a-G-T'
14 seq.partition("aa") # ('ACGtCCAgTnAG', 'aa', 'GT'): a tuple
15 seq.split("aa")     # ['ACGtCCAgTnAG', 'GT']       : a list
16 '1\n2'.splitlines() # ['1', '2'] (split at line boundaries \r, \n)
```

Strings methods III

```
seq = "ACGtCCAgTnAGaaGT"

# Deleting
seq.lstrip()      # remove leading whitespace characters
seq.rstrip()      # remove trailing whitespace characters
seq.strip()       # remove whitespace characters from both ends

seq.lstrip("AC") # 'GtCCAgTnAGaaGT'   (remove C's or A's)
seq.lstrip("CA") # 'GtCCAgTnAGaaGT'   (remove C's or A's)
seq.lstrip("C")  # 'ACGtCCAgTnAGaaGT' (no impact)
# same for rstrip but from the right and strip from both ends

# Simple parsing of text lines from CSV files
line.strip().split(',') # remove newline and split CSV (\t if TSV)

# translate (case sensitive)
table = seq.maketrans('atcg', 'tagc') # map characters by index
seq.lower().translate(table)          # 'tgcaggtcantcttca'
```

Exercise

Create the following directory structure

- Dokumente
  - python
    - notebooks
    - data

Jupyter Notebook

- File: Literals.ipynb
- URL: `https://cbdm.uni-mainz.de/mb17`
- Download the file into the notebooks folder

Sequence types

Text sequence type:

- **Strings**: immutable sequences of characters

Basic sequence types:

- **Lists**: mutable sequences
- **Tuples**: immutable sequences
- **Ranges**: immutable sequence of numbers

Sequence operations:

- All sequence types support common sequence operations (slide 98)
- Mutable sequence types support specific operations (slide 99)

Lists I

A **List** is an ordered collection of objects

```
1  List1 = [] # an empty list
2
3  List1 = ['b', 'a', 1, 'cat', 'K', 'dog', 'F']
4  List1[0]   # 'b'   (access item of index 0)
5  List1[1]   # 'a'   (access item of index 1)
6  List1[-1]  # 'F'   (access the last item)
7  List1[-2]  # 'dog' (access the second last item)
8
9  # Slices [start:end:step]
10 List1[2:5]   # [1, 'cat', 'K']           (index 5 excluded)
11 List1[:5]    # ['b', 'a', 1, 'cat', 'K']
12 List1[5:]    # ['dog', 'F']
13 List1[-2:]   # ['dog', 'F']
14 List1[0:5:2] # ['b', 1, 'K']
```

Lists II

```
# Built-in functions
List2 = [1, 2, 3, 4, 5]
len(List1) # 5      (length = 7 items)
max(List2) # 5
min(List2) # 1
sum(List2) # 15

# List methods
List2 = []                # empty list
List2.append(1)           # [1]
List2.append('A')         # [1, 'A']
List2.extend(['B',2])     # [1, 'A', 'B', 2]
List2.pop(2)              # [1, 'A', 2]
List2.insert(3, 'A')      # [1, 'A', 2, 'A'] (insert 'A' at index 3)
List2.index('A')          # 1 (index of the 1st 'A')
List2.count('A')          # 2 (number of 'A')
List2.reverse()           # ['A', 2, 'A', 1]
```

Lists III

```
# sorting
List3 = [5, 3, 4, 1, 2]
sorted(List3) # [1, 2, 3, 4, 5] (build a new sorted list)
List3         # [5, 3, 4, 1, 2] (List3 not changed)
List3.sort()  # modifies the list in-place
List3         # [1, 2, 3, 4, 5] (.sort() did modify List3!)

# nested list / 2D lists / tables
myList = [ ['b', 'a' ] ,
           [ 1 , 'cat'] ] # a list of 2 lists
myList[0]       # returns the first list ['b', 'a']
myList[0][0]    # 'b' (1st item of the 1st list)
myList[0][1]    # 'a' (2nd item of the 1st list)
myList[1]       # returns the 2nd list [1, 'cat']
myList[1][0]=10 # [['b', 'a'], [10, 'cat']]
```

Lists IV

```
1  myList = [ ['b', 'a'  ] ,
2             [ 1 , 'cat'] ]
3
4  for sublist in myList:          # loop over sublists
5      for value in sublist:       # loop over values
6          print(value)            # print 1 value per line
7  # b
8  # a
9  # 10
10 # cat
11
12 for sublist in myList:                # loop over sublists
13     new_sublist = map(str, sublist) # convert each item to string
14     print('\t'.join(new_sublist))    # print as TSV table
15 # b    a
16 # 10   cat
```

Tuples and ranges

A **Tuple** is an ordered collection of objects

```
1 Tuple1 = ()                                          # empty tuple
2 Tuple1 = ('b', 'a', 1, 'cat', 'K', 'dog', 'F') # defined tuple
3
4 Tuple1[0]                    # 'b'
5 Tuple1[1:3]                  # ('a', 1) (index 3 excluded)
```

Ranges

```
1 # Range(start, stop[, step])
2 range(10)                    # range(0, 10) => no nice print method
3 list(range(10))              # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4 list(range(0, 30, 5))        # [0, 5, 10, 15, 20, 25]
5 list(range(0, -5, -1))       # [0, -1, -2, -3, -4]
6 list(range(0))               # []
7 list(range(1, 0))            # []
```

Sets I

A **Set** is a mutable unordered collection of objects

```
S0 = set()                  # an empty set
S0 = {'a', 1}               # a new set of 2 items
S1 = {'a', 1, 'b', 'R'} # a new set of 4 items
S2 = {'a', 1, 'b', 'S'} # a new set of 4 items
len(S0)                     # 2

# Operators
'R' in S1           # True
'R' not in S2       # True
S1 - S2             # in S1 but not in S2 => {'R'}
S1 | S2             # in S1 or in S2 => {1, 'a', 'S', 'R', 'b'}
S1 & S2             # in S1 and in S2 => {1, 'b', 'a'}
S1 ^ S2             # in S1 or in S2 but not in both => {'R', 'S'}
S0 <= S1            # S0 is subset of S2 => True
S1 >= S2            # S1 is superset of S2 => False
S1 >= S0            # True
S0.isdisjoint(S1) # False
```

Sets II

```
# Methods
S0.copy()          # return a new set with a shallow copy of S0
S0.add(item)       # add element item to the set
S0.remove(item)    # remove element item from the set
S0.discard(item)   # remove element item from the set if present
S0.pop()           # remove and return an arbitrary element
S0.clear()         # remove all elements from the set
```

Dictionaries I

A **Dictionary** is a mutable indexed collection of objects (indexed by unique keys)

```
d = {}                          # empty dictionary
d = {'A':"ALA", 'C':"CYS"} # dictionary with 2 items
d['A']          # 'ALA'
d['C']          # 'CYS'
d['H']="HIS" # add new item
d               # {'H': 'HIS', 'C': 'CYS', 'A': 'ALA'}
del d['A']    # {'C': 'CYS', 'H': 'HIS'}

'C' in d      # True (key 'C' is in d)
'A' not in d # True (key 'A' is not in d anymore)
```

Dictionaries II

d[key]	get value by key
d[key] = val	set value by key
del d[key]	delete item by key
d.clear()	delete all items
len(d)	number of items
d.copy()	make a shallow copy
d.keys()	return a view of all keys
d.values()	return a view of all values
d.items()	return a view of all items (key,value)
d.update(d2)	add all items from dictionary d2
d.get(key [, val])	get value by key if exists, otherwise val
d.setdefaults(key [, val])	like d.get(k,val), also set d[k]=val if k not in d
pop(key[, default])	remove key and return its value, return default otherwise.
d.popitem()	remove a random item and returns it as tuple

Table: Functions for dictionaries

Converting types I

Many Python functions are sensitive to the type of data. For example, you cannot concatenate a string with an integer:

```
1  sign = 'You are ' +     21  + '-years-old' # error !!
2  sign = 'You are ' + str(21) + '-years-old' # OK
3  sign # 'You are 21-years-old'
4
5  # convert to int (from str or float)
6  int('2014')   # from a string
7  int(3.141592) # from a float
8
9  # convert to float (from str or int)
10 float('1.99') # from a string
11 float(5)      # from an integer
```

Converting types II

```
# convert to str (from int, float, list, tuple, dict and set)
str(3.141592)  # '3.141592'
str([1,2,3,4]) # '[1, 2, 3, 4]'

# convert a sequence type to another
# (str, list, tuple, and set functions)
new_set   = set(old_list)   # list to set
new_tuple = tuple(old_list) # list to tuple
new_set   = set("Hello")    # string to set {'H','o','e','l'}
new_list  = list("Hello")   # string to list ['H','e','l','l','o']
```

Copy I

- Assignments (=) do not copy objects, they create bindings between a target and an object.

```
1  # Numeric types (immutable)
2  a = 1      # a binds the object 1
3  b = a      # b binds the object 1
4  b = b + 1 # b binds a new object created by the sum
5  a          # 1
6  b          # 2
7
8  # Strings (immutable)
9  a = "Hello"                          # a binds the object "Hello"
10 b = a                                # b binds the object "Hello"
11 a = a.replace('o', 'o World!') # a binds a new object
12 a                                    # 'Hello World!'
13 b                                    # 'Hello'
```

Copy II

- For collections that are mutable or contain mutable items, a **shallow copy** is sometimes needed so one can change one copy without changing the other.

```
# Dictionary (mutable)
d1 = {'A':"ALA", 'C':"CYS"} # d1 binds the object
d2 = d1                     # d2 binds the object
d2['H']="HIS" # add item to the object
d1            # {'A': 'ALA', 'H': 'HIS', 'C': 'CYS'}
d2            # {'A': 'ALA', 'H': 'HIS', 'C': 'CYS'}

d2 = d1.copy() # d2 binds a shallow copy of the object
d2['P']="PRO" # add item to the copied object
d1          # {'A': 'ALA', 'H': 'HIS', 'C': 'CYS'}
d2          # {'A': 'ALA', 'H': 'HIS', 'P': 'PRO', 'C': 'CYS'}
```

Copy III

```
1  # List (mutable)
2  l1 = ['A', 'H', 'C']
3  l2 = l1
4  l2.append('P')
5  l1 # ['A', 'H', 'C', 'P']
6  l2 # ['A', 'H', 'C', 'P']
7
8  l2 = l1[:] # shallow copy by assigning a slice of the all list
9  l2.append('V')
10 l1 # ['A', 'H', 'C', 'P']
11 l2 # ['A', 'H', 'C', 'P', 'V']
```

Copy IV

- Convert types to get copies

```
1 new_list  = list(oldlist)  # shallow copy
2 new_dict  = dict(olddict)  # shallow copy
3 new_set   = set(oldlist)   # copy list as a set
4 new_tuple = tuple(oldlist) # copy list a tuple
```

- The copy module

```
1 import copy
2 x.copy()     # shallow copy of x
3 x.deepcopy() # deep copy of x, including embedded objects
```

For loop I

```
1  # For items in a list
2  for person in ['Isabel', 'Kate', 'Michael']:
3    print ("Hi", person)
4  # Hi Isabel
5  # Hi Kate
6  # Hi Michael
7
8  # For items in a dictionary
9  seq = ''                          # an empty string
10 d = {'A':"ALA", 'C':"CYS"} # a dictionary with 2 keys
11 for k in d.keys():                # loop over the keys
12   seq += d[k]                     # append value to seq
13 print(seq)                        # 'CYSALA'
```

For loop II

```
1  # For items in a string
2  for c in 'abc':
3    print (c)
4  # a
5  # b
6  # c
7
8  # For items in a range
9  for n in range(3):
10     print(n)
11 # 0
12 # 1
13 # 2
14
15 # For items from any iterator
16 for n in iterator:
17     print(n)
```

Enumerate

```
# loop getting index and value
RNAs = ['miRNA', 'tRNA', 'mRNA']
for i, rna in enumerate(RNAs):
    print(i, rna)
# 0 miRNA
# 1 tRNA
# 2 mRNA

# loop over 2 lists
RNAtypes = ['micro', 'transfer', 'messenger']
for i, t in enumerate(RNAtypes):
    r = RNAs[i]
    print(i, t, r)
# 0 micro miRNA
# 1 transfer tRNA
# 2 messenger mRNA
```

While loop

```
1  i=0
2  value=1
3  while value<200:
4      i+=1
5      value *= i
6      print (i, value)
7  # 1 1
8  # 2 2
9  # 3 6
10 # 4 24
11 # 5 120
12 # 6 720
```

Exercise

URL

- `https://cbdm.uni-mainz.de/mb17`

Jupyter Notebook

- File: Sequences.ipynb
- Download the file into the notebooks folder

Data file

- File: shrub_dimensions.csv
- Download the file into the data folder

Functions I

```
1  from random import choice  # import function 'choice'
2
3  # Simple function
4  def kmerFixed():           # define function kmerFixed
5   print("ACGTAGACGC")       # print predefined string
6
7  kmerFixed()                # display 'ACGTAGACGC'
8
9  # Returning a value
10 def kmer10():              # define function kmer10
11  seq=""                    # define an empty string
12  for count in range(10):   # repeat 10 times
13      seq += choice("CGTA") # add 1 random nt to string
14  return(seq)               # return string
15
16 newKmer = kmer10()         # get result of function into variable
17 print(newKmer)             # call the function e.g. 'ACGGATACGC'
```

Functions II

```
1  # One parameter
2  def kmer(k):                    # define kmer with 1 param. k
3      seq=""
4      for count in range(k): # k is used to define the range
5          seq+=choice("CGTA")
6      return(seq)
7
8  print(kmer(k=4)) # e.g. 'TACC'
9  print(kmer(20))  # e.g. 'CACAATGGGTACCCCGGACC'
10 print(kmer(0))   #
11 print(kmer())    # TypeError: kmer() missing 1 required
12                  #            positional argument: 'k'
```

Functions III

```
# Parameters with more parameters and default values
def generic_kmer(alphabet="ACGT", k=10):
    seq=""
    for count in range(k):
        seq+=choice(alphabet)
    return(seq)

generic_kmer("AB12", 15) # e.g. '112AA1A12AA1121'
generic_kmer("AB12")     # e.g. '1AA1B1BA2A'
generic_kmer(k=20)       # e.g. 'GTGGGCTTGTGCCCTGCACT'
generic_kmer()           # e.g. 'CTTGCCGGGA'
generic_kmer(k=8, alphabet="#$%&") # e.g. '$$#&%$%$'
```

Name spaces I

- Variable and function names defined globally can be seen in functions: this is the global namespace

```
1 a = 10                  # global variable
2
3 def my_function():
4     print(a)            # will use the global variable
5
6 my_function()           # 10 (the global a)
7 print(a)                # 10 (the global a)
```

Name spaces II

- Names defined within a function can not be seen outside: the function has its own namespace.

```
1  a = 10                  # global variable
2
3  def my_function():
4      a = 1               # local variable defined by assignment
5      b = 2               # local variable defined by assignment
6      print(a)
7
8  my_function()           # 1  (the local a)
9  print(a)                # 10 (the global a)
10 print(b)                # NameError: name 'b' is not defined
```

Name spaces III

- Use parameters and returned values to get and set variables outside the name space

```
1  a = 10                    # global variable
2
3  def my_function(val): # local variable val
4      b = 2
5      val = val + b
6      return(val)
7  print(a)                  # 10 (the global a)
8  print(my_function(a)) # 12
9  print(a)                  # 10 (the global a unchanged)
10
11 c = my_function(a)     # set val to 10 and assign 10+2 to c
12 print(c)                  # 12 (global a was changed)
13 print(a)                  # 10 (global a was unchanged)
14
15 a = my_function(a)     # change global a with value 10+2
```

Truth Value Testing I

Any object can be tested for truth value. The following values are considered false (other values are considered True):

- None
- False
- zero value: e.g. 0 or 0.0
- an empty sequence or mapping: e.g. ' ', (), [ ], { }.

Operations and built-in functions that have a Boolean result always return 0 for False and 1 for True

Boolean Operations I

A Boolean is equal to True or False

- a and b (true if a and b are true, false otherwise)
- a or b (true if a or b is true (1 alone or both), false otherwise)
- a ^ b (true if either a or b is true (not both), false otherwise)
- not b (true if b is false, false otherwise)

Boolean Operations II

All example code for tests below return ”True” unless otherwise specified

```
1  # let set values of 3 variables (single "=" symbol)
2  a = True
3  b = False
4  c = True
5
6
7  # simple tests using two "=" symbols (==)
8  a == True
9  b == False
10 c == True
```

Boolean Operations III

```
1  # let set values of 3 variables (one "=" symbol)
2  a = True
3  b = False
4  c = True
5
6  # order is irrelevant
7  (a or b) == (b or a)
8  (a and b) == (b and a)
9
10 # neutral (whatever value of a)
11 (a or False) == a
12 (a and True) == a
13
14 # always the same (whatever value of a)
15 (a and False) == False
16 (a or True)   == True
```

Boolean Operations IV

```
1  # let set values of 3 variables (one "=" symbol)
2  a = True
3  b = False
4  c = True
5
6  # precedence "==" > "not" > "and" > "or"
7  (a and b or c) == ((a and b) or c)
8  (not a == b) == (not (a == b))
9
10 # equivalent expressions
11 ((a or b) or c) == (a or (b or c)) == (a or b or c)
12 (a or a or a) == a
13 (b and b and b) == b
14
15 b and b and b == b # False and False and True => False!!
16
17 a and (b or c) == (a and b) or (a and c)
18 a or (b and c) == (a or b) and (a or c)
```

Comparisons

```
Operations
<                    # strictly less than
<=                   # less than or equal
>                    # strictly greater than
>=                   # greater than or equal
==                   # equal (two symbols =)
math.isclose(a, b)   # equal for floating points a and b
!=                   # not equal
is                   # object identity
is not               # negated object identity
x < y <= z           # is equivalent to "x < y and y <= z"
```

- Comparisons between objects of same class are supported if operator defined for the class.
- Different numerical types can be compared: e.g. 2<4.56
- Floating points can not be compared exactly due to the limited precision to represent infinite numbers such as 1/3 = 0.33333...

Conditionals

- IF-ELIF-ELSE

```
1  seq = 'ATGAnnATG'
2  if 'n' in seq:
3    print ("sequence contains undefined bases (n)")
4  elif 'x' in seq:
5     print ("sequence contains unknown bases x but not n")
6  else:
7    print ("no undefined bases in sequence")
8
9  #
10 # sequence contains undefined bases
```

- ELIF and ELSE are optional
- multiple ELIF are possible

Exercise

URL

- `https://cbdm.uni-mainz.de/mb17`

Jupyter Notebook

- File: Conditionals.ipynb
- Download the file into the notebooks folder

RE: Regular Expressions I

- Regular expressions (called REs, or regexes, or regex patterns) are a powerful language for matching text patterns (**re** module)
- In Python a regular expression search is typically written as:

```
1 match = re.search(expression, string)
```

- The **re.search()** method takes a regular expression pattern and a string and searches for that pattern within the string.
- If the search is successful, **re.search()** returns a **Match object** (actually class '_sre.SRE_Match') or **None** otherwise.

RE: Regular Expressions II

```
1 import re                                          # import re module
2 str = 'an example word:cat!!'                      # Example string
3 match = re.search(r'word:\w\w\w', str) # Search a pattern
4 if match:
5     print('found', match.group())                  # 'found word:cat'
6 else:
7     print('did not find')
```

- In the pattern string, \w codes a character (letter, digit or underscore)
- The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change.

RE: Basic Patterns

Pattern	Match
a, X, 9, <	ordinary characters match themselves exactly
.	a period matches any single character except newline
\w	matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]
\W	matches any non-word character
\b	boundary between word and non-word
\s	a single whitespace character – space, newline, return, tab, form [\n \r \t \f]
\S	matches any non-whitespace character
\t	tab
\n	newline
\r	return
\d	decimal digit [0-9]
^	circumflex (top hat) matches the start of a string
$	dollar matches the end of a string
\	inhibits the "specialness" of a character. So, for example, use \. to match a period

Table: Regular expressions: basic patterns

RE: Basic examples I

The basic rules of RE search for a pattern within a string are:

- The search proceeds through the string from start to end, stopping at the first match found
- All of the pattern must be matched, but not all of the string
- If **match = re.search(pat, str)** is successful, match is not **None** and in particular **match.group()** is the matching text

RE: Basic examples II

```
1  match = re.search(r'iii', 'piiig') # found
2  match.group() == "iii"             # True
3
4  match = re.search(r'igs', 'piiig') # not found
5  match == None                      # True
6
7  match = re.search(r'..g', 'piiig') # found
8  match.group() == "iig"             # True
9
10 match = re.search(r'\d\d\d', 'p123g') # found
11 match.group() == "123"                # True
12
13 match = re.search(r'\w\w\w', '@@abcd!!') # found
14 match.group() == "abc"                   # True
```

RE: Repetitions I

Repetitions are defined using +, *, ? and { }

- + means 1 or more occurrences of the pattern to its left
  - e.g. i+ = one or more i's
- * means 0 or more occurrences of the pattern to its left
- ? means match 0 or 1 occurrences of the pattern to its left
- curly brackets are used to specify exact number of repetitions
  - e.g. A{5} for 5 A letters
  - A{6,10} for 6 to 10 A letters

Leftmost and Largest:

- First the search finds the leftmost match for the pattern, and second it tries to use up as much of the string as possible
- i.e. + and * go as far as possible (they are said to be "greedy").

RE: Repetitions II

```
# simple repetitions
re.search(r'pi+',   'piiig'   ).group() # piii
re.search(r'pi?',   'ap'      ).group() # p
re.search(r'pi?',   'apii'    ).group() # pi
re.search(r'pi*',   'ap'      ).group() # p
re.search(r'pi*',   'apii'    ).group() # pii
re.search(r'pi{3}', 'apiiiii' ).group() # piii
re.search(r'i+',    'piigiiii').group() # ii (1st hit only)

# 3 digits possibly separated by whitespaces (\s*)
re.search(r'\d\s*\d\s*\d', 'xx1 2   3xx').group() # "1 2   3"
re.search(r'\d\s*\d\s*\d', 'xx12  3xx'  ).group() # "12  3"
re.search(r'\d\s*\d\s*\d', 'xx123xx'    ).group() # "123"
```

RE: Sets of characters I

- Square brackets indicate a set of characters
  - [ABC] matches 'A' or 'B' or 'C'.
- The codes \w, \s etc. work inside square brackets too with the one exception that dot (.) just means a literal dot
- Dash indicate a range or itself if put at the end
  - [a-z] for lowercase alphabetic characters
  - [a-zA-Z] for alphabetic characters
  - [AB-] for A, B or dash
- Circumflex (^) at the start inverts the set
  - [^AB] for any character except A or B.

RE: Sets of characters II

```
1 str = 'purple alice-b@google.com monkey dishwasher'
2 match = re.search(r'\w+@\w+', str)
3 if match:
4     print match.group()  ## 'b@google'
5
6 match = re.search(r'[\w.-]+@[\w.-]+', str)
7 if match:
8     print match.group()  ## 'alice-b@google.com'
```

RE: Functions I

RE module functions:

- **re.match()** returns a Match object if occurrence found at begining of string, None otherwise
- **re.search()** returns a Match object for 1st occurrence, None if not found
- **re.findall()** returns a list of matched sub strings, an empty list if not found
- **re.finditer()** returns an iterator on Match objects of the occurrences, an empty iterator if not found

Match object methods:

- **match.start()** returns start index
- **match.end()** returns end index
- **match.span()** returns start and end index in a tuple
- **match.group()** returns matched string

RE: Functions II

```
1  import re
2  seq = "RPAPPDRAPDQX" # A sequence
3  expr = 'A.{1,2}D'    # A and D separated by 1 or 2 characters
4
5  match = re.search(expr, seq)
6  if match:
7      print(
8          match.start(),                      # start index
9          match.end(),                        # end index
10         match.span(),                       # start and end index
11         match.group(),                      # the matched string
12         seq[match.start():match.end()], # the matched string
13         sep=' - '
14     )
15 # 2 - 6 - (2, 6) - APPD - APPD
```

RE: Functions III

```python
import re
seq = "RPAPPDRAPDQX" # A sequence
expr = 'A.{1,2}D'    # A and D separated by 1 or 2 characters

match = re.match(expr, seq)      # Not found at begining
print(match)
# None

matches = re.findall(expr, seq) # Found 2 occurrences
print(matches)
# ['APPD', 'APD']

matches = re.finditer(expr, seq) # Found 2 occurrences
for m in matches:                 # Iterate over Match objects
    print( m.span(), m.group() ) # Use each Match object
# (2, 6) APPD
# (7, 10) APD
```

RE: Group Extraction

- Groups are defined with parentheses
- On a successful search
  - match.group(): the whole match text
  - match.group(1): match text of 1st left parenthesis
  - match.group(2): match text of 2nd left parenthesis
  - ...

```
1 import re
2 str = 'purple alice-b@google.com monkey dishwasher'
3 match = re.search('([\w.-]+)@([\w.-]+)', str)
4 if match:
5     print(match.group())   ## 'alice-b@google.com'
6     print(match.group(1))  ## 'alice-b'
7     print(match.group(2))  ## 'google.com'
```

RE: Group Extraction and Findall

- If the pattern includes a single set of parenthesis, then **findall()** returns a list of strings corresponding to that single group
- If the pattern includes 2 or more parenthesis groups, then instead of returning a list of strings, **findall()** returns a list of **tuples**. Each tuple represents one match of the pattern, and inside the tuple is the **group(1)**, **group(2)** ... data.

```
1 str = 'alice@google.com, monkey bob@abc.com dishwasher'
2 tuples = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
3 print(tuples)
4 # [('alice', 'google.com'), ('bob', 'abc.com')]
5
6 for t in tuples:
7     print(t[0], t[1], sep=' | ')
8 # alice | google.com
9 # bob | abc.com
```

RE: Options

The re functions take options to modify the behavior of the pattern match. The option flag is added as an extra argument to the **search()** or **findall()** etc., e.g. **re.search(pat, str, re.IGNORECASE)**.

- IGNORECASE ignores upper/lowercase differences for matching
- DOTALL allows dot (.) to match newline – normally it matches anything but newline.
  - Note that \s (whitespace) includes newlines
- MULTILINE allows ^and $ to match the start and end of each line within a string made of many lines. Normally they just match the start and end of the whole string.

Greedy vs. Non-Greedy

- .* or .+ return the largest match (aka it is ”greedy”)
- to get nested occurrences use .*? or .+?

```
1 string = '<b>foo</b> and <i>so on</i>' # string with xml tags
2
3 matches = re.findall(r'<.*>', string)              # <.*>
4 print(matches) # ['<b>foo</b> and <i>so on</i>'] # got all string
5
6 matches = re.findall(r'<.*?>', string)            # <.*?>
7 print(matches) # ['<b>', '</b>', '<i>', '</i>'] # got each tag
```

Substitution

- **re.sub(expression, replacement, string)**

```
1 text1 = 'alice@google.com and bob@abc.net'
2 text2 = re.sub(r'\.\w+', r'.de', text1)
3 print (text2)
4 # alice@google.de and bob@abc.de
```

- \1, \2 ... in replacement refer to match group(1), group(2) ...

```
1 text1 = 'alice@google.com and bob@abc.com'
2 text2 = re.sub(
3     r'([\w\.-]+)@([\w\.-]+)', # Expression
4     r'\2@\1',                 # Replacement string
5     str)                      # Input string
6 print (text2)
7 ## google.com@alice and abc.com@bob
```

Exercise

URL

- `https://cbdm.uni-mainz.de/mb17`

Jupyter Notebook

- File: Regex.ipynb
- Download the file into the notebooks folder

Data file

- File: sequences.tsv
- Download the file into the data folder

References

- Python documentation
  - https://docs.python.org
- Online tutorials (Python 2 or 3)
  - Google's Python Class
  - ProgrammingForBiologists.org

Escape sequences

Escape Sequence	Meaning
\newline	Backslash and newline ignored
\\	Backslash (\)
\’	Single quote (’)
\”	Double quote (”)
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Table: Escape sequences

Common Sequence Operations

Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n or n * s	equivalent to adding s to itself n times
s[i]	ith item of s, origin 0
s[i:j]	slice of s from i to j
s[i:j:k]	slice of s from i to j with step k
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s
s.index(x[, i[, j]])	index of the first occurrence of x in s (at or after index i and before index j)
s.count(x)	total number of occurrences of x in s

Table: Sequence operations sorted in ascending priority. s and t are sequences of the same type, n, i, j and k are integers and x is an arbitrary object that meets any type and value restrictions imposed by s.

Operations on mutable sequence types

Operation	Result
s[i] = x	item i of s is replaced by x
s[i:j] = t	slice of s from i to j is replaced by the contents of the iterable t
del s[i:j]	same as s[i:j] = []
s[i:j:k] = t	the elements of s[i:j:k] are replaced by those of t
del s[i:j:k]	removes the elements of s[i:j:k] from the list
s.append(x)	appends x to the end of the sequence (same as s[len(s):len(s)] = [x])
s.clear()	removes all items from s (same as del s[:])
s.copy()	creates a shallow copy of s (same as s[:])
s.extend(t) or s += t	extends s with the contents of t (for the most part the same as s[len(s):len(s)] = t)
s *= n	updates s with its contents repeated n times
s.insert(i, x)	inserts x into s at the index given by i (same as s[i:i] = [x])
s.pop([i])	retrieves the item at i and also removes it from s
s.remove(x)	remove the first item from s where s[i] == x
s.reverse()	reverses the items of s in place

Table: s is an instance of a mutable sequence type, t is any iterable object and x is an arbitrary object that meets any type and value restrictions imposed by s

Built-in functions

abs()	Return the absolute value of a number.
all()	Return True if all elements of the iterable are true (or if the iterable is empty).
any()	Return True if any element of the iterable is true. If the iterable is empty, return False.
ascii()	Return a string containing a printable representation of an object (escape non-ASCII characters).
bin()	Convert an integer number to a binary string.
bool()	Convert a value to a Boolean.
chr()	Return the string representing a character.
dict()	Create a new dictionary.
dir()	Return the list of names in the current local scope.
float()	Convert a string or a number to floating point.
format()	Convert a value to a ”formatted” representation.
help()	Invoke the built-in help system.
hex()	Convert an integer number to a hexadecimal string.

Table: Python built-in functions