Bioinformatical problem solving with Python

Wednesdays 17:30-19:00, M801

alexander.nater@uni-konstanz.de

Dictionaries

- A dictionary is an unordered container of key/value pairs.
- Keys have to be unique string objects.
- my_dict = {} or my_dict = dict() to create new empty dict
- my_dict = {“a”: 1, “b”: 2, “c”: 3}
- my_dict[“key”]=value

Dictionaries

-
  ```
  for key in my_dict:
      do something with key …
  ```

-
  ```
  for value in my_dict.values():
      do something with value …
  ```

-
  ```
  for key, value my_dict.items():
      do something with key and value …
  ```

Example

- Count how often each word in some random text occurs. Obtain a list of words appearing more than once.

Example

```
string = “””Some random text … “””
word_counts = {}
for word in string.replace(‘.’, ‘’).split():
    if word in word_count:
        word_count[word] += 1
    else:
        word_count = 1

not_unique = []
for word, count in word_counts.items():
    if count > 1:
        not_unique.append(word)
```

Functions

- Python offers a large set of built-in functions, e.g. sum(), len(), type(), help(), etc.
- We can define our own functions.
- Functions are objects and can be passed to other functions, stored in lists, etc.
- Functions may take arguments (input) and may return a value (output).

Functions

```
def get_gc_content(dna):
    “””Takes a string and calculates GC content.”””
    length = len(dna)
    g_count = dna.count(‘G’)
    c_count = dna.count(‘C’)
    gc_content = (g_count + c_count) / length
    return gc_content
```

Exercise

- Write a function to calculate the sum of all elements in a list.

- Write a function to calculate Fibonacci numbers, taking a starting and a end value for the sequence.

Exercise

- Write a function to calculate the sum of all elements in a list.

```
def get_sum(my_list):
    total = 0
    for value in my_list:
        total += value
    return total
```

Exercise

- Write a function to calculate Fibonacci numbers, taking a starting and a end value for the sequence.

```
def get_fibonacci(start, end):
    first, second = 0, 1
    fibs = []
    if start == 0:
        fibs.append(0)
    while second <= end:
        if second >= start:
            fibs.append(second)
        first, second = second, first + second
    return fibs
```

Functions

- Functions can have default values:

```
def get_fibonacci(start, end=100)
get_fibonacci(10, 1000)
get_fibonacci(10)
get_fibonacci() -> error!
```

- Functions can be called with named arguments:

```
get_fibonacci(end=1000, start=10)
get_fibonacci(end=1000, 10) -> error!
```

Encapsulation

- Used to avoid repeating commonly used pieces of code.

- Functions are completely independent from the state of the program.

- Enhance maintainability, since changes don’t affect calling code as long as interface (arguments and return value) remains unchanged.

- Variables defined in functions are only visible from within the function.

Exercise

```
def get_gc_content(dna):
    “””Takes a string and calculates GC content.”””
    length = len(dna)
    g_count = dna.count(‘G’)
    c_count = dna.count(‘C’)
    gc_content = (g_count + c_count) / length
    return gc_content
```

- What could be potential problems with the use of the get_gc_content function?
- Improve the get_gc_content function to address these issues.

Exercise

```
def get_gc_content(dna):
    “””Takes a string and calculates GC content.”””
    dna = dna.upper() # convert string to upper case
    g_count = dna.count(‘G’)
    c_count = dna.count(‘C’)
    a_count = dna.count(‘A’)
    t_count = dna.count(‘T’)
    gc_content = (g_count + c_count) / sum([g_count,
c_count, a_count, t_count])
    return gc_content
```

Lambda functions

- Get list of squared values from list:

```
my_list = [1,2,3,4,5,6]
squared_list = []
for value in my_list:
    squared_list.append(value**2)
```

- Better use built-in function ‘map’:

```
def get_squared(x):
    return x**2
squared _list = list(map(get_squared, my_list))
```

- Using lambda function to avoid defining a function:

```
squared_list = list(map(lambda x: x**2, my_list))
```

Exercise

- Write a function to get the factorial of non-negative integer n:
  n! = n * (n-1) * (n-2) * (n-3) … * 1

```
def get_factorial(n):
    rv = n
    for i in range(1, n):
        rv *= i
    return rv
```

Recursion

- A function can call itself from within the function body.

- Recursive functions are useful to solve problems that depend on solving smaller instances of the same problem.

- n! = n * (n-1)!

-
  ```
  def get_factorial(n):
      if n==1:
          return 1
      else:
          return n * get_factorial(n-1)
  ```

Exercise

- Write a recursive function to calculate the sum of the first n positive integers:
  sum = 1 + 2 + 3 + 4 + 5 …

-
  ```
  def get_sum(n):
      if n==1:
          return 1
      else:
          return n + get_sum(n-1)
  ```