Python for Language Processing

(4a) Namespaces

Dr. Jakob Prange

Fakultät für Angewandte Informatik - Universität Augsburg

CL Fall School 24

Credit: This course is based on material developed by
Annemarie Friedrich, Stefan Thater, Michaela Regneri, and Marc Schulder at Saarland University

Notebook

You can work along with the slides and try out the examples in `4b_Lambda.ipynb`.

Recap: Functions

Functions are subprograms that can (and should) be used to divide a larger problem into several smaller problems.

```
1  def factorial(x):
2  '''Computes the factorial of x'''
3    r = 1
4    for i in range(x):
5      r *= (i + 1)
6    return r
7
8  y = factorial(4)
```

Function Application

The `range(x)` function returns an iterator over integers `[0, ..., x-1]`.

```
1  def factorial(x):
2    """Computes the factorial of x"""
3    r = 1
4    for i in range(x):
5      r *= (i + 1)
6    return r
7
8  y = factorial(4)
```

- **Call by reference** (Python, Java): When the funtion is called, a reference (pointer) to the object is passed to the function.
- The function call evaluates to the value returned by the function.

In-Class Exercise: Sorting

Implement a naive sorting algorithm (do not use `sorted`, `sort`, but you can use: `pop, append, range`).

```
1  def naive_sort(x):
2    """Create new list. Find smallest value
3    in list, append to new list, remove
4    from current list. SUBPROBLEM? """
5    pass
6
7  if __name__ == '__main__':
8      # test
9      my_list = [5, 3, 1, 4, 8, 2, 7, 6]
10     my_sorted_list = naive_sort(my_list)
11     print(my_sorted_list)
12         # should print: [1, 2, 3, 4, 5, 6, 7, 8]
```

Local Variables

```
1  def factorial(x):
2    """Computes the factorial of x"""
3    r = 1
4    for i in range(x):
5      r *= (i+1)
6    return r
7  y = factorial(4)
```

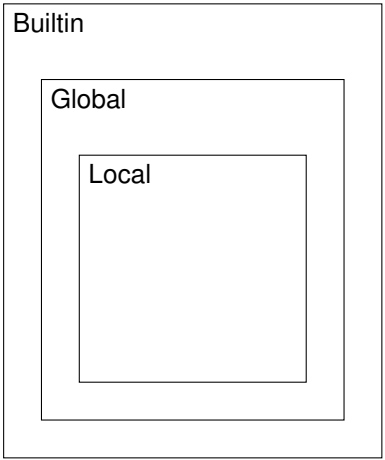
- Functions introduce local variables: parameters and variables to which a value is assigned within the scope of the function.
  $\Rightarrow$ In this case: `x`, `i`, and `r`.
- Local variables are not visible outside the function.

Variables and Namespaces

- Variables live in **namespaces**: namespaces map variables to their values.
- Namespaces can be nested: Function calls create local namespaces, which are embedded within the namespace of the calling context.
- Variables with the same name in different namespaces can refer to different values: Local variables "shadow" non-local (global) variables.

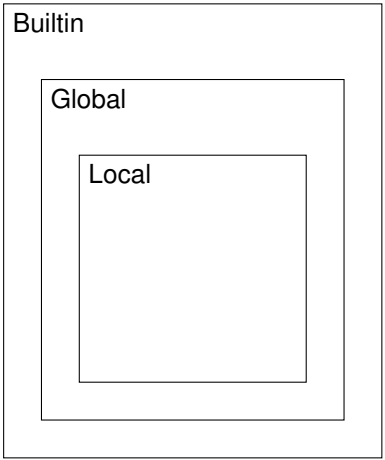
```
1  r = 'something'
2
3  def factorial(x):
4      r = 1
5      for i in range(x):
6          r *= (i + 1)
7      return r
8
9  y = factorial(4)
10 print(y, r)
11 # prints: 24 something
```

Variables and Namespaces



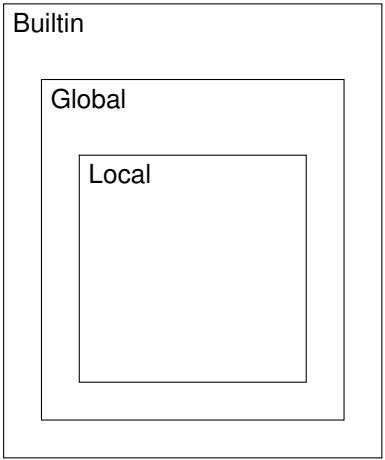
- **Builtin** namespace
  - created when Python starts
  - contains built-in names (e.g., `abs` function or type/class names)
- **Global** namespace
  - created when the program is executed (read in)
  - contains the top-level names
- **Local** namespace
  - created when a function is called
  - contains the local variables

Variables and Namespaces



- **Builtin** namespace
  - created when Python starts
  - contains built-in names (e.g., `abs` function or type/class names)
- **Global** namespace
  - created when the program is executed (read in)
  - contains the top-level names
- **Local** namespace
  - created when a function is called
  - contains the local variables

Variables and Namespaces



- **Builtin** namespace
  - created when Python starts
  - contains built-in names (e.g., `abs` function or type/class names)
- **Global** namespace
  - created when the program is executed (read in)
  - contains the top-level names
- **Local** namespace
  - created when a function is called
  - contains the local variables

In-Class Exercise: Variables and Namespaces

Which names/variables are part of which namespace?

```
1  # Identify the namespaces.
2  x = type(5) == int
3
4  def f(x):
5      x += 10
6      for i in range(5):
7          x += 1
8      return x
9
10 y = f(3)
11 print(x, y)
```

```
1  # What happens here?
2  x = type(5) == int
3
4  def f(x):
5      x += 10
6      for x in range(5):
7          x += 2
8      return x
9
10 y = f(3)
11 print(x, y)
```

Local and Global Variables

```
1  n = 123
2
3  def add_to_n(m):
4      # value of n is
5      # read from above
6      return n + m
7
8  print(add_to_n(1))
9  # prints 124
```

```
1  n = 123
2
3  def add_to_n(m):
4      # 1. n is a local variable because
5      # we assign something to it
6      n = n + m
7      # 2. n is a local variable but yet
8      # without a value
9
10 add_to_n(1)
11 UnboundLocalError: local variable
12 'n' referenced before assignment
```

- Within a function, we can access (read the value of) global (non-local) variables.
- But we cannot assign values to a non-local variable.

Local and Global Variables

```
1  n = 123
2
3  def add_to_n(m):
4      # value of n is
5      # read from above
6      return n + m
7
8  print(add_to_n(1))
9  # prints 124
```

```
1  n = 123
2
3  def add_to_n(m):
4      # 1. n is a local variable because
5      # we assign something to it
6      n = n + m
7      # 2. n is a local variable but yet
8      # without a value
9
10 add_to_n(1)
11 UnboundLocalError: local variable
12 'n' referenced before assignment
```

- Within a function, we can access (read the value of) global (non-local) variables.
- But we cannot assign values to a non-local variable.

Local and Global Variables

```
1  n = [123, 456, 789]
2
3  def add_to_n(m):
4    # n is global variable from above
5    n[0] += m
6
7  add_to_n(1)
8  print(n)
9  # prints [124, 456, 789]
```

- Modifying the value of a non-local variable of a mutable type is possible.
- These **side effects** could be intended (we will talk about this when we get to OOP) - but often this is a source of bugs!
- **Style guide**: (1) Try to avoid side effects.
  (2) Functions with side effects should not return a value.

In-Class Exercise: Side Effects

What is printed in each case? In which case does a side effect occur?

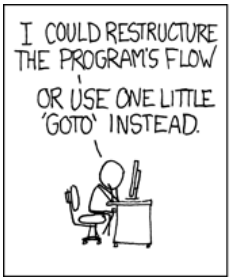
```
1  def incr(items):
2    for i in range(len(items)):
3      items[i] += 1
4    return items
5
6  example = [1, 2, 3, 4]
7  print(example)
8  print(incr(example))
9  print(example)
```

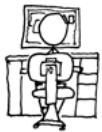
```
1  def incr(n):
2    n += 1
3
4  example = 1
5  print(example)
6  incr(example)
7  print(example)
```

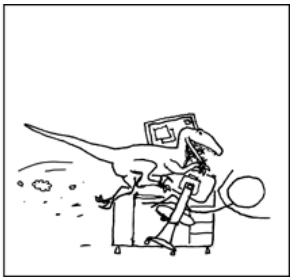
Self-Study: The `global` keyword

Work through the following tutorial to understand the `global` keyword:
https://www.programiz.com/python-programming/global-keyword

Recommendation: Use this with extreme care - makes code often hard to read.
Instead: use constants whose names are never used as local variables.







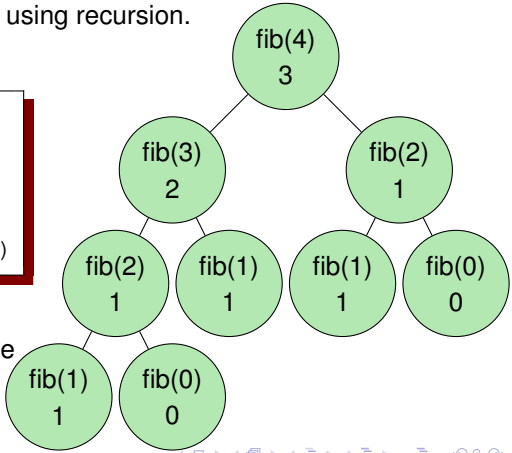
xkcd, CC-BY-NC 2.5

Recursion

- Functions can call other functions
- Functions can call themselves ⇒ This is called recursion
- Many problems can be elegantly solved using recursion.

```
1  def fib(n):
2    if n <= 1:
3      return n
4    else:
5      return fib(n-1) + fib(n-2)
```

Example: Fibonacci sequence
Each number is found by adding up the two numbers before it:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



In-Class Exercise: Recursion

1. Write a recursive function computing the factorial of a non-negative number.
   factorial(0) = 1 factorial(n) = n * factorial(n-1)
2. Implement a version of `sum()` that computes the sum of numbers in a possibly nested list of lists.

```
>>> nestedsum([1, 2, 3, 4, 5])
15
>>> nestedsum([1, [2, 3, [4], []], [5]])
15
```

Functions inside Functions

```
1  def outer(x):
2    def inner(y):
3      return x + y
4    return inner(1)
```

- Often used to implement helper functions that perform some of the computations of another function.
- What does `outer(7)` return?

Functions inside Functions

```
1  def outer(x):
2    def inner(y):
3      return x + y
4    return inner
5
6  f1 = outer(1)
7  f2 = outer(7)
8  print(f1(3))
9  print(f1(4))
```

- Functions are also **objects** in Python.
- Functions can return other functions.
- What are the values of `f1` and `f2`? What is printed? Try it out and explain.

Keyword Arguments

```
 1  def sqrt(x, precision=.00001):
 2    """Computes the square root of x"""
 3    g = x
 4    while (g * g) - x > precision:
 5      g = (g + x/g) / 2
 6    return g
 7
 8  # ---------------------------------#
 9  >>> sqrt(2)
10  1.4142156862745097
11  >>> sqrt(2, precision = .01)
12  1.4166666666666665
13  >>> sqrt(2, .01)
14  1.4166666666666665
```

Keyword Arguments

- The **default value** is evaluated when the function definition is evaluated (read in).
- This can have strange effects when the default value is a list (or some other value that can be modified).

```
 1  def f(someparameter = []):
 2    someparameter.append(1)
 3    return someparameter
 4
 5  print(f())      # [1]
 6  print(f())      # [1, 1]
 7  print(f([]))    # [1]
 8
 9  # What happens here?
10  x = []
11  print(f(x))
12  print(f(x))
```

Exercise: Keyword Arguments

Read on more details on keyword arguments, e.g., using this excellent tutorial:
https://www.geeksforgeeks.org/default-arguments-in-python

1. How can we fix the function `f` such that it actually creates a default empty list when called without a value for `someparameter`?
2. Which of the following cases are valid? Which ones will result in errors? Why?

```
1  def f(x, y="Hello", z=5):
2    return str(x) + y + str(z)
3
4  f(27, y="Bonjour")
5  f(y="Salut", 101)
6  f(42, z=1)
7  f(s="Bye")
8  f()
```

Positional and Keyword Arguments

- Positional arguments: need to come first, no “name” given when calling the function, order matters!
- Python is quite flexible: `*positional` gives access to all positional arguments (as a tuple).
- `**keywords` gives access to a dictionary with all keyword-value pairs.
- Can also be combined with regular arguments: Try it out!

```
1  def f(*positional, **keywords):
2    print("Positional", positional)
3    print("Keywords", keywords)
4
5  f(1, 2, 3)
6  f(a=1, b=2)
7  f(1, 2, a=2)
```

Function Annotations

- PEP 3107: Values of function arguments and return values can be specified.
- These are just annotations, i.e., Python does not enforce these types!
  (Intention: use with external software, e.g., for documentation generation.)
- Can increase code readability.

```
1  def f(x : list, y : int = 0) -> str:
2    x.append(y)
3    s = ""
4    for i in x:
5      s += str(i)
6    return s
7
8  print(f([1, 2]))
```

Self-Study: Anonymous Functions

Work through `4b_Lambda.ipynb`.

Self-Study: Docstring Conventions

The Python PEP 0257 proposes a standard way how to define docstrings. This way of documenting your code is highly recommended. Read on the conventions and use this style from now on to document your homework.

```
1  def complex(real=0.0, imag=0.0):
2      """Form a complex number.
3
4      Keyword arguments:
5      real -- the real part (default 0.0)
6      imag -- the imaginary part (default 0.0)
7      """
8      if imag == 0.0 and real == 0.0:
9          return complex_zero
10     ...
```