Programmieren I (Python)

Christian Osendorfer

2023-11-02

Higher Order Functions

Higher Order Functions

What are higher-order functions?

A function that either:

- Takes another function as an argument
- Returns a function as its result

All other functions are considered *first-order* functions.

Generalizing over computational processes

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

The *common structure* among these functions may be a **computational process**, not just a number.

Functions as arguments

```
1  def cube(k):
2      return k ** 3
3
4  def summation(n, term):
5      """Sum the first N terms of a sequence.
6      >>> summation(5, cube)
7      225
8      """
9      total = 0
10     k = 1
11     while k <= n:
12         total = total + term(k)
13         k = k + 1
14     return total
```

Functions as return values

Locally defined functions

Functions defined within other function bodies are bound to names in a local frame (more on **frames** later in the course.)

```
def make_adder(n):
    """Return a function that takes one argument k
       and returns k + n.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

Lambda Expressions

A lambda expression is a simple **function definition that *evaluates* to a function**.

The syntax:

```
lambda <parameters>: <expression>
```

A function that takes in parameters and returns the result of expression.

A lambda version of the square function:

```
square = lambda x: x * x
```

- A function that takes in parameter `x` and returns the result of `x * x`.

Lambda syntax tips

A lambda expression does not contain return statements or **any statements at all**.

Incorrect:

```
1 square = lambda x: return x * x
```

Correct:

```
1 square = lambda x: x * x
```

Def statements vs. Lambda expressions

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the `def` statement gives the function an intrinsic name, which is available to be called by that name from a python interpreter (see **frames** later in this course).

Lambda as argument

It’s convenient to use a lambda expression when you are passing in a simple function as an argument to another function.

Instead of…

```
1 def cube(k):
2     return k ** 3
3
4 summation(5, cube)
```

… we can use a `lambda`:

```
1 summation(5, lambda k: k ** 3)
```

Conditional Expressions

Conditional expressions

A conditional expression has the form:

```
<consequent> if <predicate> else <alternative>
```

Evaluation rule:

- Evaluate the `predicate` expression.
- If it's a `True` value, the value of the whole expression is the value of the .
- Otherwise, the value of the whole expression is the value of the .

Lambdas with conditionals

This is invalid syntax:

```
1 lambda x: if x > 0: x else: 0
```

Conditional expressions to the rescue!

```
1 lambda x: x if x > 0 else 0
```

Recursion

Recursive functions

A function is recursive if the body of that function calls itself, either directly or indirectly.

Recursive functions often operate on increasingly smaller instances of a problem.

The problems within the problem

- The sum of the digits of 6 is simply 6.
- Generally: the sum of any one-digit non-negative number is that number.
- The sum of the digits of `2021` is the sum of `202` plus `1`.
- Generally: the sum of a number is the sum of the first digits (`number // 10`), plus the last digit (`number % 10`).

Summing digits without a loop

```
1  def sum_digits(n):
2      """Return the sum of the digits of positive integer n.
3      >>> sum_digits(6)
4      6
5      >>> sum_digits(2021)
6      5
7      """
8      if n < 10:
9          return n
10     else:
11         all_but_last = n // 10
12         last = n % 10
13         return sum_digits(all_but_last) + last
```

Anatomy of a recursive function

- **Base case**: Evaluated without a recursive call (the smallest subproblem).
- **Recursive case**: Evaluated with a recursive call (breaking down the problem further).
- **Conditional statement** to decide if it’s a base case.

Recursive factorial

The factorial (*Fakultät*) of a natural number n is defined as:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```python
def fact(n):
    """
    >>> fact(0)
    1
    >>> fact(4)
    24
    """
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Tree Recursion

Tree Recursion

Tree-shaped processes arise whenever a recursive function makes more than one recursive call (*multiple recursion*).

Sierpinski curve

Recursive Virahanka-Fibonacci

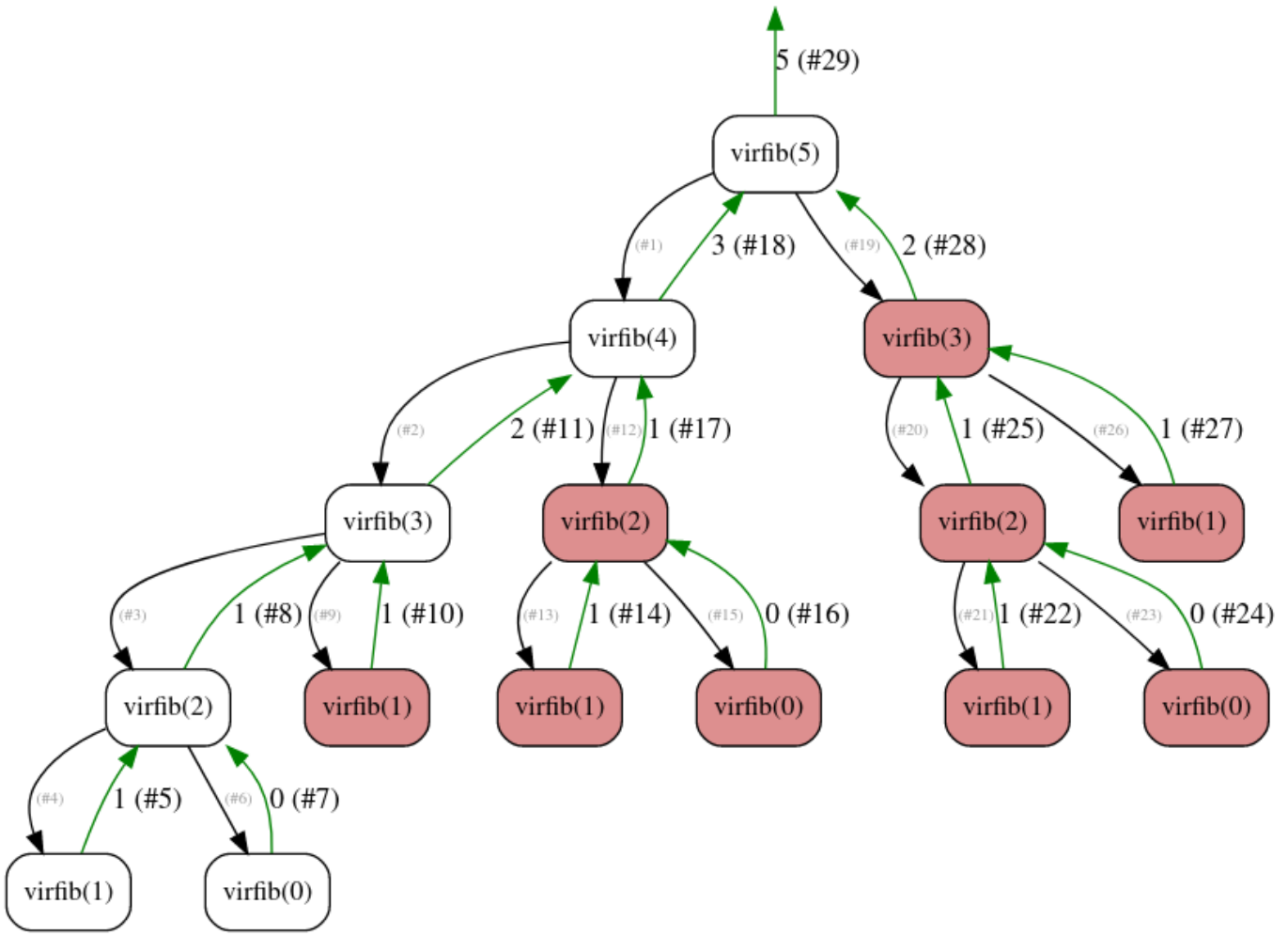
The nth number is defined as:

$$\mathrm{vf(n)} = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{vf}(n-1) + \mathrm{vf}(n-2) & \text{otherwise} \end{cases}$$

```python
def virfib(n):
    """Compute the nth Virahanka-Fibonacci number, for n >= 1.
    >>> virfib(2)
    1
    >>> virfib(6)
    8
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return virfib(n-1) + virfib(n-2)
```

Redundant computations

The function is called on the same number multiple times.



Object Oriented Programming (OOP)

Objects

- Python supports many different kinds of data (`123`, `2.4`, `Hi`, `[1,2]`).
- Each *is an object* and every object has
  - a type.
  - an internal data representation (primitive or *composite*).
  - a set of procedures to interact with an object.
- An object is *an instance of a type*.

OOP

- **Everything in python is an object** (and has a type).
- **Create new objects** of some type.
- **Manipulate objects**.
- **Destroy objects**.
  - explicitly using `del` or just “forget” about them.
  - python system will reclaim destroyed or inaccessible objects – **garbage collection**.

What are objects

- Objects are a data abstraction that captures
  - an internal representation
    - through **data attributes**.
  - an interface for interacting with object
    - through **methods**.
  - defines behaviors but **hides** implementation.

Advantages of OOP

- Bundle data into packages together with procedures that work on them through well-defined interfaces.
- Divide-and-conquer development
  - Implement and test behavior of each class separately.
  - Increased modularity reduces complexity.
- **Classes** make it easy to reuse code
  - Many Python modules define new classes.
  - Each class has a separate environment (no collision on function names).
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior.

Create vs Use a class

- Make a distinction between creating a class and using an **instance** of the class.
- Creating the class involves
  - Defining the class name
  - Defining class attributes
- Using the class involves
  - Creating new instances of objects (using a **constructor**).
  - Doing operations on the instances.

Defining your own types

- Use the `class` keyword to define a new type

```
1 class Coordinate(object):
2 #define attributes here
```

- Similar to `def`, **indent code** to indicate which statements are part of the class definition.
- The word `object` means that Coordinate is a Python object and **inherits** all its attributes.
  - `Coordinate` is a **subclass of `object`**.
  - `object` is a **superclass of `Coordinate`**.

What are attributes ?

- Data and procedures that “belong” to the class.
- Data attributes
  - think of data as other objects that make up the class.
  - for example, a coordinate is made up of two numbers.
- Methods (procedural attributes)
  - Think of methods as functions that only work with this class.
  - How to interact with the object?
  - for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects.

Creating an instance of a class

- How can we create an **instance** of a class?
- Use a specialized method called `__init__` to *initialize* some data attributes.

```
1 class Coordinate(object):
2   def __init__(self,, x, y):
3     self.x = x
4     self.y = y
```

Creating an instance of a class

- `Coordinate(x=1, y=2)` is often called a **constructor**.
- When the constructor is called:
  - A new instance of that class is created
  - The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

Instance variables

- Instance variables are data attributes that describe the state of an object.
- This `__init__` initializes 4 instance variables:

```
1 class Product:
2
3     def __init__(self, name, price, nutrition_info):
4         self.name = name
5         self.price = price
6         self.nutrition_info = nutrition_info
7         self.inventory = 0
```

- The object's methods can then change the values of those variables or assign new variables.