Procedural Programming

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

1 Programming paradigms

2 Procedural programming
- What is a function
- Variable scope
- Function calls and parameter passing

Programming paradigms

What are programming paradigms?

A way to classify programming languages, or programs, based on their features

- Most programming languages support more than one paradigm
- Many programming languages use multiple paradigms in their implementation
- Widely use paradigms are the imperative (via procedural and object-oriented programming) and declarative (via functional and logic programming)

Programming paradigms

Programming paradigms 101

https://cs.lmu.edu/~ray/notes/paradigms/

Programming paradigms for dummies – what every programmer should know (Peter van Roy)

https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf

Paradigms and the relations between them (photo in article)

https://en.wikipedia.org/wiki/Programming_paradigm

Paradigms supported by well known languages

https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

Procedural programming

- **Imperative programming** describes computation in terms of statements that change a program state.
- In **procedural programming**, programs are assembled from a set of subroutines (or procedures, or functions) that talk to one another via input and return parameters.
- In our understanding, writing functions is **not enough** to implement procedural programming!

Procedural programming

How to implement procedural programming (in A5 and beyond)

- Use functions as an interface to access and modify the representation of domain entities (*setters* and *getters*)
- Pass the information functions need to do their job as input parameters
- Return the result of the computation, or signal that an error happened using return codes
- Replace global variables with local variables that are sent as function parameters
- Functions should either handle the user interface (use input/print), or they should work using parameters (don't use input/print)

What is a function

A self contained block of statements that:

- Has a *name*,
- May have a list of (formal) *parameters*,
- May *return* a value
- Has a specification which consists of:
  - A *short description*
  - *Type and description of parameters*
  - Conditions imposed over input parameters (*precondition*)
  - Type and description for the return value
  - Conditions that must be true after execution (*post-condition*).
  - Any Exceptions raised

What is a function

```python
def maximum(x,y):
    """
    Return the maximum of two values
    input: x,y - the parameters to compare
    output: The largest of the parameters
    Error: TypeError - parameters cannot be compared
    """
    if x > y:
        return x
    return y
```

What is a function

- Can you tell what the function below does?
- Did it take more than a few seconds?

```
def f(c):
    b = []
    while not sol(b) and c != []:
        cand = next(c)
        c.remove(cand)
        if acceptable(b + [cand]):
            b.append(cand)
    if sol(b):
        found(b)
    return None
```

NB!

A function without specification is not complete!

What is a function

Every non-trivial, non-UI function should:

- Use meaningful names (function name, parameter and variable names)
- Provide specification
- Include comments
- Have a test function (will come later)

What is a function

```python
def greedy(c):
    '''
    Generic greedy algorithm
    input: c - set of candidates
    output: solution of generic problem
    '''
    # The empty set is the candidate solution
    b = []
    while not solution(b) and c != []:
        # Select best candidate (local optimum)
        candidate = selectMostPromising(c)
        c.remove(candidate)
        # If the candidate is acceptable, add it
        if acceptable(b + [candidate]):
            b.append(candidate)
    if solution(b):
        return b
    # In case no solution
    return None
```

What is a function

- A **function definition** is an executable statement introduced using the keyword **def**.
- The function definition does not execute the function body; this gets executed only when the function is called. A function definition defines a user-defined function object.

```python
def maximum(x,y):
    """
    Return the maximum of two values
    input: x,y - the parameters to compare
    output: The largest of the parameters
    Error: TypeError - parameters cannot be compared
    """
    if x > y:
        return x
    return y
```

Variable scope

The *scope* defines a name' visibility within a block. If a local variable is defined in a block, its scope includes that block. All variables defined at a particular indentation level or scope are considered local to that indentation level or scope

Variable scope

Uncomment each code section in **ex24_VariableScope.py**. Figure out what happens and why.

Variable scope - the LEGB rule

Python uses the **L**ocal, **E**nclosing, **G**lobal and **B**uilt-in (**LEGB**) rules for scoping

- **L**ocal scope - The body of the function where the name was defined; each function call creates a new scope (including recursively)
- **E**nclosing scope - In case of nested functions, names in the outer scope are visible in the inner one
- **G**lobal scope - Names defined at the module's top level (e.g., "global variables")
- **B**uilt-in scope - Names built into Python (e.g., built-in functions[1]), they are avaialble when running the program

Name lookup: Local ►Enclosing ►Global ►Builtin ►**Error** ☹

[1] https://docs.python.org/3/library/functions.html

Variable scope - useful functions

- **locals()** - Update and return a dictionary representing the current local symbol table[2]
- **globals()** - Return the dictionary implementing the current module namespace[3]
- **vars()** - Return the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute[4]
- **dir()** - Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object[5]

[2] https://docs.python.org/3/library/functions.html#locals

[3] https://docs.python.org/3/library/functions.html#globals

[4] https://docs.python.org/3/library/functions.html#vars

[5] https://docs.python.org/3/library/functions.html#dir

Variable scope

Python scope, the LEGB rule and useful functions

https://realpython.com/python-scope-legb-rule/#using-scope-related-built-in-functions

Calls

- A **block** is a part of the program that is executed as a unit. In Python, blocks of code are denoted by line indentation
- A **function body** is a block. A block is executed in an *execution frame*. When a function is invoked a new execution frame is created
- A new execution frame is created for each recursive call!

Execution frames

`http://www.pythontutor.com/visualize.html`

Some more details here

`https://medium.com/@marcosanchezayala/the-python-tutor-1adc76be5ff1`

Calls

An execution frame contains:

- Some administrative information (used for debugging)
- Determines where and how execution continues after the code block's execution has completed
- Defines two namespaces, the local and the global namespace, that affect execution of the code block (**locals()** and **globals()** dictionaries)
- A *namespace* is a mapping from names (identifiers) to objects. A particular namespace may be referenced by more than one execution frame, and from other places as well.

Calls

- Adding a name to a namespace is called binding a name (to an object); changing the mapping of a name is called rebinding.
- Removing a name is unbinding.
- Namespaces are functionally equivalent to dictionaries (and often implemented as dictionaries).

Discussion

What did the output of locals(), globals() look like?

Calls

Function call visualisation

Enter the code in example **ex25_function_call_visualisation.py** into `https://pythontutor.com/visualize.html` and run it step by step

- Check the order in which the recursive calls are made
- Each call creates a new execution frame
- Actual calculation is done when functions return from the call stack

Parameter passing - important concepts

- **Formal parameter** - an identifier for an input parameter of a function. Each call to the function must supply a corresponding value (argument) for each mandatory parameter
- **Actual parameter** - a value provided by the caller of the function for a formal parameter
- The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called (arguments are passed *by object reference*, or *by assignment*)

Parameter passing - important concepts

- **Pass by value** - the argument is evaluated, and a copy of the evaluation result is bound to the formal parameter of the function
- **Pass by reference** - function receives a reference to the actual argument, rather than a copy to its value
- **Side effect** - a function that modifies the caller's environment (beside producing a value) is said to have side effects

Parameter passing - in practice

Parameter passing

ex26_parameter_passing.py

Side Effects

ex27_side_effects.py

To better understand what happens

`https://medium.com/school-of-code/passing-by-assignment-in-python-7c829a2df10a`

- **TLDR;** Object references are passed by value

Parameter passing - in practice

Discussion

What are the advantages and disadvantages of pass by value and pass by reference?