Процедурное программирование

Егор Суворов

Курс «Парадигмы и языки программирования», группа 18.Б09-пу

Среда, 20 февраля 2019 года

Историческая справка

«Подпрограммы» — выделили кусок кода, дали имя, можно вызывать.

```
arr = [1, 2, 3]
def print_array():
    print(' '.join(map(str, arr)))
print_array()
arr.append(3)
print_array()
```

Отличие от goto — можно вызвать из любого места:

- Можно перед вызовом дописать в конец goto куда надо
- Можно честный стек вызовов, тогда допустима рекурсия

Получаем механизм *абстракции* для кода (раньше были только примитивы и композиция).

Параметры

Так неудобно:

```
vector<int> a, b, c;  // Длинная арифметика
void subtract() {
        // c = a - b;
}
// b = c - a;  // ???
```

Так уже лучше:

```
void subtract(const vector<int> &a, const vector<int> &b,
              vector<int> &result) {
        // c = a - b;
}
vector<int> a, b, c;  // Длинная арифметика
subtract(a, b, c);  // c = a - b;
```

Теперь полностью контролируем *данные*, с которыми работает код

Возвращаемое значение

Конвенция для шаблона:

```
void add(int a, int b, int *result) {
    *result = a + b;
}
int x; add(2, 2, &x);
```

Теперь так:

```
int add(int a, int b) {
    return result;
}
int x = add(2, 2);
```

- Имеет смысл только если в языке есть выражения
- В разных языках параметры и возвращаемые значения реализованы по-разному $\Rightarrow$ разные ограничения
- В языках вроде C/C++/Rust есть ещё разные *конвенции вызовов* о том, как передавать параметры на ассемблере

Python: множественный возврат

Идиома: возвращаем кортеж, а на месте распаковываем:

```
def partition(arr, middle):
    return ([x for x in arr if x < middle],
            [x for x in arr if x >= middle])
left, right = partition([1, 2, 3, 4, 1], 3)
```

Скобки можно опустить:

```
def divmod(a, b):  # Встроенная функция
    return a // b, a % b
q, r = divmod(10, 3)
```

Осторожно: если значений много, то легко перепутать порядок или типы; для сложных вещей лучше вернуть объект (будут дальше)
В C++ тоже есть (см. `pair<iterator, bool> std::map::insert`), но не так популярно.

Python: именованные параметры

При вызове можно говорить, какой параметр чему равен:

```
def line_contains(a, b, c, x, y):
        return a * x + b * y == c
print(line_contains(1, 1, 2, 1, -1))
print(line_contains(a=1, b=1, c=2, x=1, y=-1))
```

- Читать код проще: не надо помнить порядок
- Какие-то аргументы можно не указать
- Можно указать аргументы в другом порядке

```
assert int('13', 8) == 11
assert int('13', base=8) == 11
```

В C++ такого нет: имя параметра при компиляции стирается.

Python: необязательные параметры

Можно задать значение по умолчанию:

```
def foo(fname, what=None, why=None):
    what = what or 'somewhat'
    why = why or 'somwhy'
    print(fname, what, why)
foo('foo.txt')  # foo.txt somewhat somewhy
foo('foo.txt', what='x')  # foo.txt x somewhy
foo('foo.txt', why='x')  # foo.txt somewhat x
foo('foo.txt', 'x')  # foo.txt x somewhy

open('file.txt', 'r')
open('file.txt', 'r', 1024, 'utf-8')
open('file.txt', encoding='utf-8', mode='r', buffering=1024)
```

Python: значения необязательных параметров

```
def append_1(lst=[]):
    lst.append(1)
    return lst
print(append_1([1, 2, 3]))  # [1, 2, 3, 1]
print(append_1([]))  # [1]
print(append_1([]))  # [1]
print(append_1())  # [1]
print(append_1())  # [1, 1]
print(append_1([]))  # [1]
print(append_1())  # [1, 1, 1]
```

Причина: значение по умолчанию «вшивается» в функцию в момент её создания, а не каждый раз при вызове.

Решение: использовать `None` и `or`.

Python: строго именованные параметры

Если знаем, что пользователи запутаются в порядке аргументов:

```
def foo(fname, *, what=None, why=None):
    what = what or 'somewhat'
    why = why or 'somwhy'
    print(fname, what, why)
foo('foo.txt')  # foo.txt somewhat somewhy
foo('foo.txt', what='x')  # foo.txt x somewhy
foo('foo.txt', why='x')  # foo.txt somewhat x
foo('foo.txt', 'x')  # Error
```

Python: произвольные параметры

Можно попросить сложить все неизвестные позиционные параметры в кортеж (одна звёздочка):

```
def foo(a, b, *args, c=123):
    print(a, b, c, args)
foo(1, 2, 3, 4, 5, 'foo', c=10)
# 1 2 10 (3, 4, 5, 'foo')
```

Можно попросить сложить все именованные параметры в словарь (две звёздочки):

```
def bar(*args, **kwargs):
    print(args, kwargs)

bar(10, 20, 30, a=1, b=2)
# (10, 20, 30) {'a': 1, 'b': 2}
```

Python: распаковка в аргументы

Python — язык динамический, тут можно код на ходу генерировать и изменять:

```
def bar(*args, **kwargs):
    print(args, kwargs)
bar(*(10, 20, 30), *(40, 50, 60), **{'a': 1, 'b': 2})
# (10, 20, 30, 40, 50, 60) {'a': 1, 'b': 1}


print(*[1, 2, 3, 4])
print(1, 2, 3, 4)
```

В C++ такое сделать сложновато.

Когда выделять процедуру

- Код встречается много раз по всей программе
- Код хочется протестировать отдельно
- Хочется абстрагироваться от конкретного способа решения подзадачи

Продвинутые вопросы:

1. Насколько легко пользоваться такой процедурой правильно (легко)? А неправильно по ошибке (сложно)?
2. Если не выносить в процедуру, действительно ли потребуется менять все части кода согласованно?
3. Сколько кода экономит вынесение в процедуру?
4. Какие решения мы принимаем за пользователя процедуры?
5. Насколько часто процедура будет использоваться?
6. Что случится, если в процедуре баг и повлияет на всю программу?

На демо посмотрим

Время для демо!

Стандартные куски

Выделяйте в разные процедуры:

- Чтение данных из внешнего мира
- Вывод данных во внешний мир
- Перевод из одного представления данных в другое
- Разные стадии обработки
- Всё, что придётся отлаживать отдельно

Пример последовательности процедур для суммы на отрезке:

1. Считать массив
2. Считать параметры для генерации случайных запросов
3. **Посчитать частичные суммы, получить отдельный массив**
4. Для каждого запроса в цикле:
   1. **Сгенерировать запрос**
   2. **Посчитать ответ для запроса**
   3. Вывести ответ на экран

Функции высшего порядка: резюме

- *Функция высшего порядка* — функция, которая в качестве одного из аргументов ожидает функцию.
- Элемент функционального программирования
- Ещё один способ композиции кода
- В разных языках удобен в разной степени (на Си неудобно)
- *Лямбда-функция* — функция без имени

Декораторы: резюме

Синтаксический сахар Python для добавления свойств и кода к функциям:

- Запомнить результат вызова и не пересчитывать
- Вывести на экран количество вызовов, время работы
- Перед запуском функции проверить права пользователя
- Автоматически переписать числа Фибоначчи на возведение матрицы в степень: habr.com/ru/post/236689

Стал возможным, так как функции — *объект первого класса*, с ними можно работать, как с данными.

Замыкания и область видимости: резюме

Область видимости определяется *лексически* (lexical scoping)

- Область видимости — либо глобальная, либо какая-то функция
- Каждая переменная либо «объявлена» в области видимости, либо нет (есть оператор вроде = или += или нет)
- Это определяется во время компиляции
- Каждое имя обращается к какой-то переменной в какой-то области видимости (правило LEGB[1])
- Чтобы не объявлять новую переменную: `global` и `nonlocal`
- Что-то подобное — во всех современных языках

Ещё бывает динамическая область видимости (dynamical scoping):

- От структуры кода ничего не зависит, смотрим на стэк вызовов
- Примеры: Lisp, LaTeX, Perl, bash

[1] PEP 3104

Исключения

- *Один из* способов обработки ошибок в программах
- Для *исключительных* ситуаций, когда не сразу ясно, что делать
- Идея: если нам неясно, что делать, попросим разобраться вызывающего: файл не найден $\Rightarrow$ сообщить пользователю
- Есть в половине современных языков программирования
- Обычно сильно медленнее возврата значения и не так красиво

Обработка ошибок — это сложно: надо очень хорошо понять, что делать при каких ошибках [2]. Разным задачам — разные инструменты.

- Передавать код ошибки в глобальной переменной: забывают
- Функция возвращает код ошибки: неудобно возвращать значение
- Функция возвращает либо код ошибки, либо значение. Дальше либо явно обработать (Go), либо есть сахар (Haskell, Rust)
- Перезапустить процесс (Erlang)

[2] «Открытие файла» иногда надо попробовать снова, иногда попробовать другой файл, иногда сообщить пользователю

Заключение

Классный курс по Python в Computer Science Center:
compscicenter.ru/courses/python/2015-autumn
В домашнем задании надо сделать утилиту с нуля, правильно разбить на процедуры, всё протестировать.