Утверждения assert

Для чего нужны утверждения

На практике абсолютно все программы содержат ошибки, разница лишь в том, насколько часто они проявляются и насколько критичны их последствия. При этом важным становится **обнаружить ошибку как можно скорее**.

Утверждения — это один из самых быстрых способов для обнаружения ошибок.

С другой стороны, они документируют внутреннюю работу программы и повышают её ремонтопригодность. Они делают предположения программиста, создающего код, *явными* для всех, кто читает его код.

Формы оператора assert

Оператор assert имеет две формы. Первая, более простая форма:

```
assert condition
```

где condition — это логическое выражение, которое, по вашему мнению, будет верным в момент выполнения. Если во время выполнения кода оно окажется ложным, система выбросит исключение. Проверяя, что логическое выражение на самом деле верно, инструкция `assert` подтверждает ваши ожидания от программы, увеличивая уверенность в том, что код не содержит ошибок.

Программный эквивалент:

```
if __debug__:
    if not condition:
        raise AssertionError()
```

Вторая форма утверждения — это:

```
assert condition, message
```

Эта форма позволяет при сбое отобразить в сообщении состояние ключевых переменных, из-за которых и мог произойти сбой.

Её программный эквивалент:

```
if __debug__:
    if not condition:
        raise AssertionError(message)
```

Как и все необработанные исключения, сбои утверждений содержат трассировку стека (stack trace) с именем файла и номером строки, из которой они были брошены. Часто этого достаточно для диагностики ошибки.

Внедрение утверждений в код

Пример для изолированной функции:

```
def fibonacci(n):
    assert n >= 0
    F = [0, 1] + [0]*n
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    assert F[n] >= 0 and F[n] == F[n-1] + F[n-2]
    return F[n]
```

Пример для конструктора класса:

```
class DateType:
    def __init__(self, year=2000, month=1, day=1):
        assert year >= 0 and year < 3000
        assert month >= 1 and month <= 12 and day >= 1
        assert day <= [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31][month-1]
        self.year = year
        self.month = month
        self.day = day
```

Отключение утверждений

В некоторых случаях вычисление условия может быть слишком долгим. Например, проверка того, что массив действительно отсортирован, может занимать сравнимое время с самой сортировкой. В таком случае assert-ы могут быть отключены: тогда они эквивалентны пустому оператору и в семантике, и в производительности.

Для этого следует запускать интерпретатор с опцией **-O**.

Адекватное использование утверждений

Оператор assert служит для отлавливания невероятных, недопустимых ситуаций, которые могут возникнуть из-за *внутренних* ошибок в программе.

Исключения `AssertionError` *никогда не следует перехватывать*! Они не должны быть использованы для отображения сообщений пользователю или обработки регулярно возникающей, "нормальной" исключительной ситуации. Их не следует использовать для индикации неверного ввода пользователя или ошибок операционной системы — для этого лучше использовать `raise AnyOtherException()`.

Их используют для проверки:

1. параметров функции на их тип и значение (preconditions)

2. разумности возвращаемого значения функции (postconditions)
3. инвариантов (invariants)

Поскольку элегантно проверять предусловия и постусловия можно при помощи PyContracts, главным адекватным вариантом использования `assert` становятся инварианты. Их проверка не менее важна для гарантии корректности кода, чем проверка предусловий и постусловий.

Виды инвариантов

Внутренний инвариант (internal invariant) — это логическое выражение, выражающее уверенность программиста в значении некоторых переменных в некоторый момент выполнения программы.

```
if x%3 == 0:
    print("Число делится на три.")
elif x%3 == 1:
    print("При делении на три остаток - один.")
else:
    assert x%3 == 2  # assert здесь является комментарием, гарантирующим истинность утверждения
    print("Остаток при делении на три - два.")
```

Из внутренних инвариантов стоит выделить **инвариант цикла** — это логическое выражение, истинное после каждого прохода тела цикла и перед началом выполнения цикла, зависящее от переменных, изменяющихся в теле цикла.

Инвариант потока выполнения (control-flow invariants) — выражает уверенность программиста в том как идёт поток выполнения. В том числе, что какой-то участок кода никогда не должен быть достигнут.

```
def foo():
    for ...:
        if ...:
            return ...
    assert False  # Поток выполнения никогда не должен достигнуть этой строки!
```

Инвариант класса (class invariant) — это семантические свойства и ограничения целостности экземпляра класса. Например, объект календарной даты никогда не может находиться в состоянии 31 апреля или 30 февраля. Объект класса красно-чёрного дерева поиска в момент вызова любого его метода, как и по окончании, должен быть сбалансирован.

Контракты PyContracts

Предусловия и постусловия удобно оформлять не через утверждения, а как контракт функции. Нам поможет декоратор contract из библиотеки PyContracts:

```
from contracts import contract
```

Проверка предусловий

Пример контракта с проверкой предусловий:

```
@contract(x='int,>=0')
def f(x):
    pass
```

В этом случае вызов функции с недопустимым значением аргумента приведёт к исключению ContractNotRespected, а трассировка стека будет сопровождена следующей полезной информацией:

```
>>> f(-2)

ContractNotRespected: Breach for argument 'x' to f().
Condition -2 >= 0 not respected
checking: >=0       for value: Instance of <class 'int'>: -2
checking: int,>=0   for value: Instance of <class 'int'>: -2
```

Или такой:

```
>>> f("Hello")

ContractNotRespected: Breach for argument 'x' to f().
Could not satisfy any of the 3 clauses in Int|np_scalar_int|np_scalar,array(int).
```

Дополнительную диагностику ошибки опустим.

Проверка постусловий

Пример контракта с проверкой результата работы функции:

```
@contract(returns='int,>=0')
def f(x):
    return x
```

В случае нарушения контракта также будет приведена диагностика:

```
>>> f(-1)

ContractNotRespected: Breach for return value of f().
Condition -1 >= 0 not respected
checking: >=0       for value: Instance of <class 'int'>: -1
checking: int,>=0   for value: Instance of <class 'int'>: -1
Variables bound in inner context:
```

Три варианта описания контракта функции

1. Через декоратор contract:

```
@contract(n='int,>=0', returns='int,>=0')
def f1(n):
    pass
```

2. Описание в документ-строке:

```
@contract
def f2(n):
    """ Function description.
        :type n: int,>=0
        :rtype: int,>=0
    """
    pass
```

3. Через аннотацию типов:

```
@contract
def f3(n:'int,>=0') -> 'int,>=0':
    pass
```

Использование декоратора — самый традиционный способ описания контракта. Аннотация типов — самый короткий способ использования PyContracts.

Язык описания контрактов PyContracts

Логическое И: если нужно проверить несколько условий, их можно просто записать через запятую:

```
@contract(x='>=0,<=1')
def f(x):
    pass
```

Логическое ИЛИ: вертикальная черта:

```
@contract(x='<0|>1')
def f(x):
    pass

@contract(x='(int|float),>=0')
def f(x):
    pass
```

Для списков возможны требования как к длине, так и к типу элементов и их значениям:

`list[length contract](elements contract)`

Примеры:

`list[>0]` — непустой список.

`list(int)` — список целых чисел, возможно пустой.

`list(int,>0)` — список положительных целых, возможно пустой.

`list[>0,<=100](int,>0,<=1000)` — непустой список из не более ста положительных целых чисел, не превышающих по значению тысячу.

Для словарей также можно ввести требования к их размеру, а также к типу ключа и/или типу значения:

`dict[length contract](key contract: value contract)`

Примеры:

`dict[>0]` — непустой словарь.

`dict(str:*)` — словарь со строками в качестве ключей и любыми типами значений.

`dict[>0](str:(int,>0))` — непустой словарь с ключами-строками и положительными целочисленными значениями.

Описание нового контракта

При помощи декоратора можно создать новый вид контракта:

```
@new_contract
def even(x):
    if x % 2 != 0:
        msg = 'The number %d is not even.' % x
        raise ValueError(msg)
```

После этого его можно использовать как и обычный:

```
@contract(x='int,even')
def foo(x):
    pass
```

Можно создать новый вид контракта и так:

```
new_contract('short_list', 'list[N],N>0,N<=10')

@contract(a='short_list')
def bubble_sort(a):
    for bypass in range(len(a)-1):
        for i in range(len(a)-1-bypass):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
```

Связывание значений различных параметров

В языке описания контрактов PyContracts используются переменные:

- строчные латинские буквы — для любых объектов
- заглавные латинские буквы — для целых чисел

Пример такой связки:

```
@contract(words='list[N](str),N>0',
          returns='list[N](>=0)')
def get_words_lengths(words):
    return [len(word) for word in words]
```

В этом примере контракт проверит не только то, что возвращается тип list, но и то, что этот список имеет ту же длину, что и переданный ей список words.

Замечания

1. Реализация программирования по контракту не входит в стандартную библиотеку Python. Нами использована библиотека PyContracts, но существуют и альтернативные библиотеки: PyDBC, Contracts for Python, Decontractors.
2. Домашняя страница библиотеки PyContracts: http://andreacensi.github.io/contracts/
3. В качестве дополнительного материала — доклад про контрактное программирование на Moscow Python.