Lab05 & hw05

NJU SICP22 TAs

Feeling good about your mid exam?

You have finished halfway ☺

Lab05

Nonlocal

- We can now edit parent frame variable!

Iterator

- Iterator is a (ordered) sequence of object, used:
  - For Enumeration: for item in iterators
  - As Counter & index: for i in range
  - …
- We can decorate a iterator, or build complex iterator on it (or some of them).
- Iterator can have non-trivial “semantics” in its “item” & “order”.
  - Non-trivial semantics needs non-trivial computation.
- Remember statements for iteration, “while”, “for”, …?

Generator

- A normal function describe a computation **for some single value**.
- **Sometimes we want a sequence of values** (use as a iterator) generating alongside specific computation. The sequence maybe infinite.
- Generator function describe how such sequence is computed one following the other.
  - Generator function generates generator (which is an iterator).
  - It is just a "function" replacing "return" with "yield".
  - The function never truly "returns" until the computation ends
    - Any time it yield a value and "give back its control", its frame and program counter is kept. – similar to "coroutine".

Lab05p1: Count

- Subject: Implement `count`, which takes in an iterator `t` and returns the number of times the value `x` appears in the first `n` elements of `t`. A value appears in a sequence of elements if it is equal to an entry in the sequence.

- Main point: familiarize the usage of iterator

```
def count(t, n, x):
    count = 0
    for _ in range(n):
        if next(t) == x:
            count += 1
    return count
```

Lab05p2: repeated

- Subject: Implement `repeated`, which takes in an iterator `t` and returns the first value in `t` that appears `k` times *continuously*.
- Main point: record task-related information along with iteration

```
def repeated(t, k):
    if k == 1:
        return next(t)
    key, cnt = next(t), 1
    while True:
        next_key = next(t)
        if key == next_key:
            cnt += 1
            if cnt == k:
                return key
        else:
            key, cnt = next_key, 1
```

Lab05p2: repeated

- Do not consume values more than you need.

Your implementation should iterate through the items in a way such that if the same iterator is passed into `repeated` twice, it should continue in the second call at the point it left off in the first. An example of this behavior is in the doctests.

Lab05p3: Scale

- Main point: decorate a iterator
  - Pay attention to underlying exception, which need to be handle right away explicitly

```
def scale(it, multiplier):
    try:
        while True:
            v = next(it)
            yield v * multiplier
    except StopIteration:
        return
```

Lab05p3: Scale

Important semantics information for compiler/interpreter. Keep this in mind!

- Main point: decorate a iterator
  - Or, you use built-in features carefully designed to dealt with iterator type
    - In effect: "eager evaluation" and "lazy evaluation".

```
def scale(it, multiplier):
    yield from [value * multiplier for value in it]
    yield from (value * multiplier for value in it)
    yield from map(lambda x: multiplier * x, it)

siter = scale(iter, 2)
print(next(siter))
```

Lab05p4: merge

- Subject:

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.

- Main point: combine multiple iterators into one

```
def merge(a, b):
    first_a, first_b = next(a), next(b)
    while True:
        if first_a == first_b:
            yield first_a
            first_a, first_b = next(a), next(b)
        elif first_a < first_b:
            yield first_a
            first_a = next(a)
        else:
            yield first_b
            first_b = next(b)
```

Lab05p5: hailstone

- Main point: define the infinite recursive computation for hailstone, yielding items alongside.

```
def hailstone(n):
    yield n
    if n == 1:
        return
    elif n % 2 == 0:
        yield from hailstone(n // 2)
    else:
        yield from hailstone(n * 3 + 1)
```

hw05 review

Hw05p1: Make withdraw

- Main point: use nonlocal instead of new frame for memorizing "balance". No calls to "make_withdraw" are in "withdraw" now!

```
def make_withdraw(balance, password):
    ...
    def withdraw(amount, pw):
        nonlocal balance
        ...
        balance -= amount
        return balance
    return withdraw
```

Hw05p2: Joint Account

- Main point: layer lambda with lambda (layer frame with frame) to record information.
- Remind you of high-order functions, please don't forget them!

```
def joint_withdraw(amount, pw):
    if pw == new_pass:
        return withdraw(amount, old_pass)
    return withdraw(amount, pw)
```

Hw05p03: Permutation

- Main point: clarify the math first & use recursive generator

```
head, tail = seq[0], seq[1:]
for perm in permutations(tail):
    for i in range(len(seq)):
        yield perm[:i] + [head] + perm[i:]
```

Hw05p04: Two sum

- Main point: relate the ideas of generator function and generator(iterator).
  - Generator has semantics for its generated sequence
    - Item's value
    - Item's order
- Complexity: find a value in list is O(n) time consuming, dict may be preferred.

```
def two_sum_list(target, lst):
    visited = {}
    for val in lst:
        ...
    return False
```

Hw05p4: Lookups

- Main point: element yield from a generator can be a lambda.
  - A "path" from root to target node: [label() | branch()]*
  - Clarify the semantics is important!
  - Also a recursive function.
    - Link parent to child edge to child's path.

```
def lookups(k, key):
    if label(k) == key:
        yield label
    for i in range(len(branches(k))):
        for lookup in lookups(branches(k)[i], key):
            yield (lambda f, i: lambda v: f(branches(v)[i]))(lookup, i)
```

Hw05p05: Remainders generator

- Main point: a generator for generator. Not difficult ☺
  - Inner generator has nothing special to any other value type.

```
def remainders_generator(m):
    def helper(n):
        ...
            yield i
        ...
    yield from [helper(i) for i in range(m)]
```