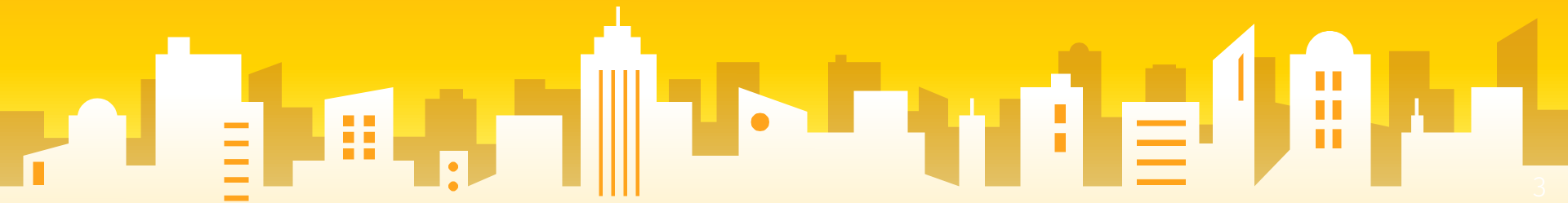
Iterators in Python

First Lesson. (Most Important)



What are iterators in Python?

- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

- Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

- An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

- The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterating Through an Iterator in Python

- We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise StopIteration.

Example

```
# define a list
my_list = [4, 7, 0, 3]
# get an iterator using iter()
my_iter = iter(my_list)
## iterate through it using next()
#prints 4
print(next(my_iter))
#prints 7
print(next(my_iter))
## next(obj) is same as obj.__next__()
#prints 0
print(my_iter.__next__())
#prints 3
print(my_iter.__next__())
## This will raise error, no items left
next(my_iter)
```

How for loop actually works?

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

Python Infinite Iterators

- It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator.

- Here is a simple example to demonstrate infinite iterators.

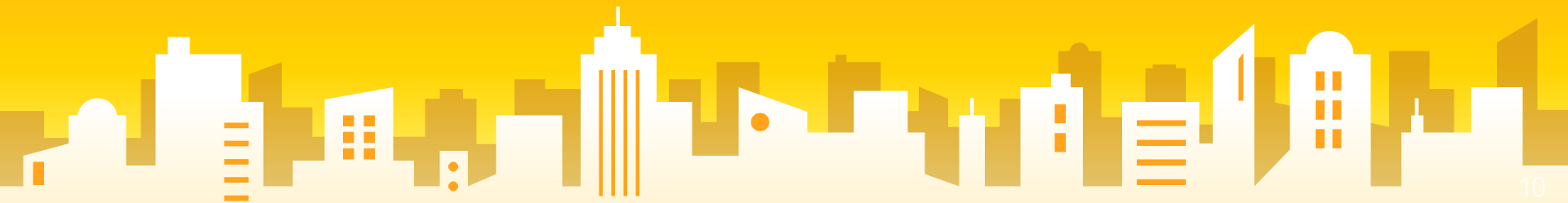
- The built-in function iter() can be called with two arguments where the first argument must be a

Example

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

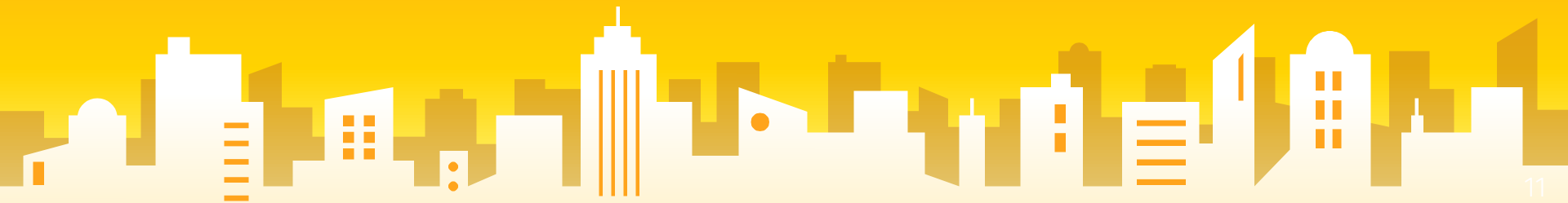
Iterating over a list

Assignment



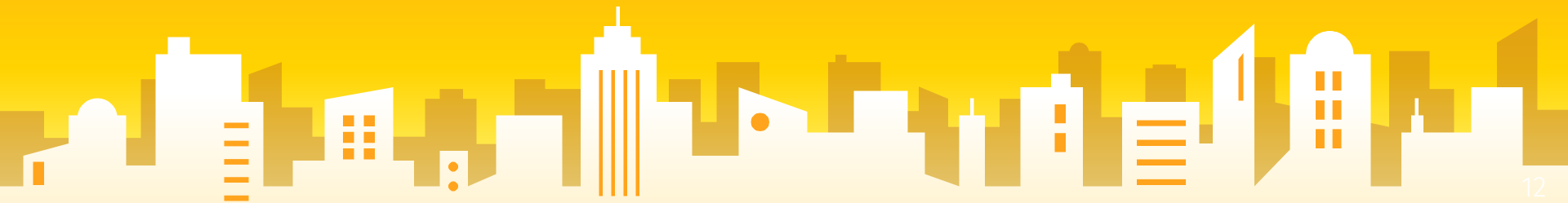
Iterating over a tuple (immutable).

Assignment



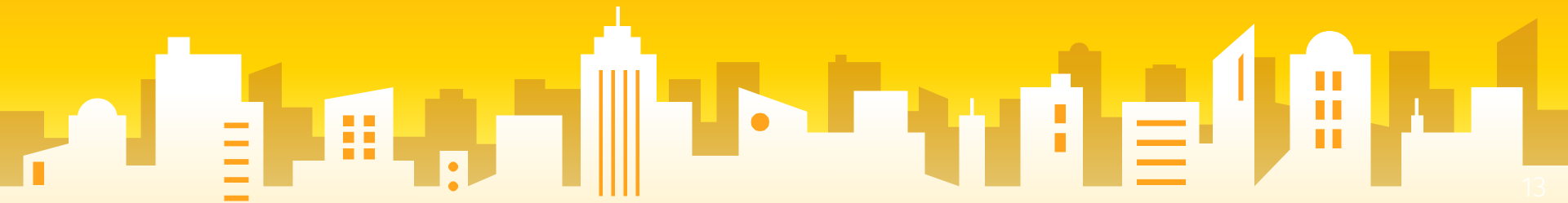
Iterating over a String.

Assignment



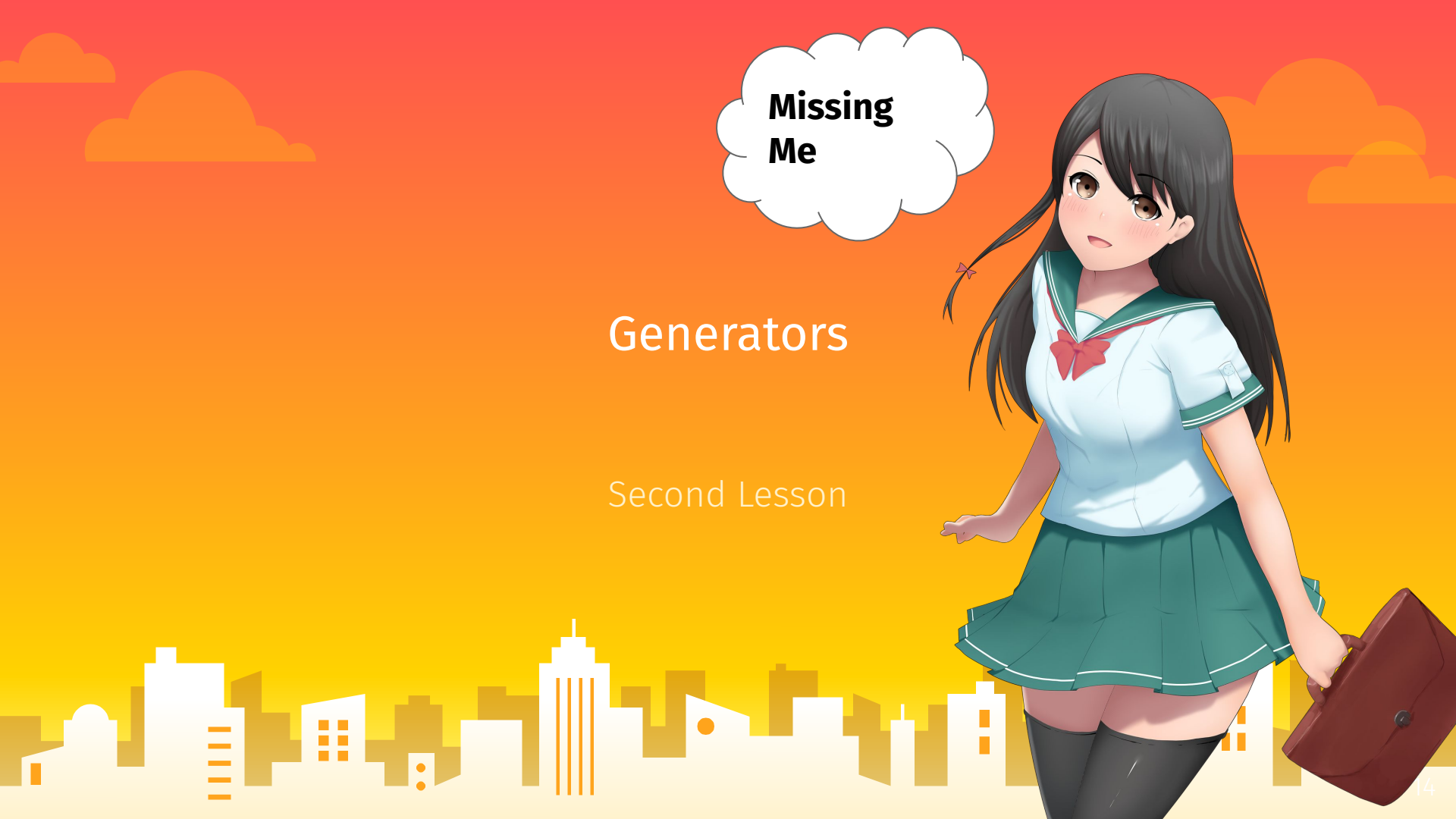
Iterating over dictionary.

Assignment



Generators

Second Lesson



What are generators in Python?

- Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python.
- Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

How to create a generator in Python?

- It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.
- If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.
- The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Example

```
for item in my_gen():
    print(item)
```

```
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n
    n += 1
    print('This is printed second')
    yield n
    n += 1
    print('This is printed at last')
    yield n
```

Explanation

- One interesting thing to note in the above example is that, the value of variable n is remembered between each call.
- Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.
- To restart the process we need to create another generator object using something like a = my_gen().

Python Generators with a Loop

- The above example is of less use and we studied it just to get an idea of what was happening in the background.
- Normally, generator functions are implemented with a loop having a suitable terminating condition.
- Let's take an example of a generator that reverses a string.

Example

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]

for char in rev_str("hello"):
    print(char)
```

Python Generator Expression

```
my_list = [1, 3, 6, 10]

[x**2 for x in my_list]

(x**2 for x in my_list)
```

Example

```
my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
next(a)
```

Sum in Generators

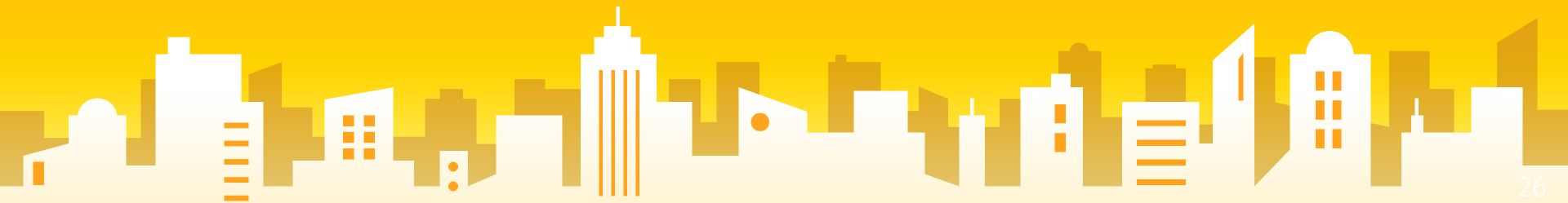
```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Why generators are used in Python?

- Easy to Implement
- Memory Efficient
- Represent Infinite Stream
- Pipelining Generators

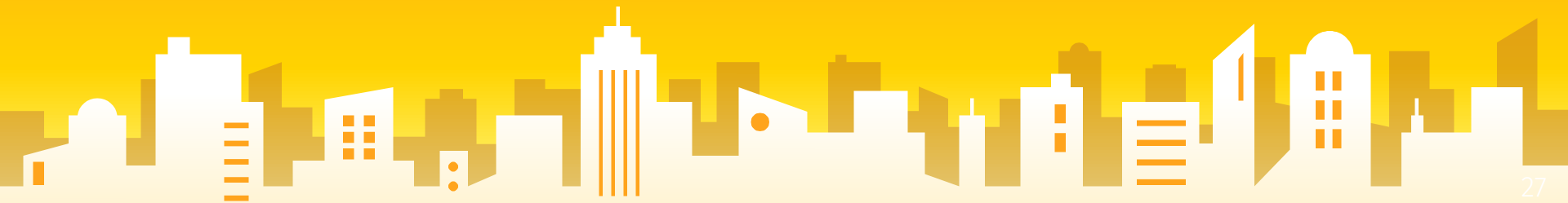
Create a countdown iterator that counts from 9 to 1. Use generator functions!

Assignment



Create a lazy filter generator function. Filter the elements of the Fibonacci sequence by keeping the even values only.

Assignment



Thank you

Miss You...