Lambda Functions

Lambda (অ্যানোনিমাস) ফাংশন — গভীর ব্যাখ্যা (বাংলা)

Lambda ফাংশন হলো ছোট, নামবিহীন (anonymous) ফাংশন যা একটি single expression দিয়ে তৈরি হয়। Python-এ এগুলো দ্রুত ছোট লজিক inline প্রকাশ করার জন্য ব্যবহার করা হয় — বিশেষ করে higher-order functions (যেমন map, filter, sorted, reduce, callback ইত্যাদি)-এর সাথে।

নীচে ধারণা থেকে শুরু করে ব্যবহার, সীমাবদ্ধতা, সতর্কতা, এবং AI/ML-স্পেসিফিক টিপস সব ধাপে দেবো।

1) মৌলিক সিনট্যাক্স

lambda arguments: expression

উদাহরণ:

```
f = lambda x, y: x + y
print(f(2, 3))  # 5
```

f একইভাবে লেখা যায়:

```
def f(x, y):
    return x + y
```

পয়েন্ট: lambda–তে *কেবল একটাই expression* থাকতে পারে — কোনো statement (like if as statement, for, while, assignment) সম্ভব নয়। (তবে ternary expression a if cond else b ব্যবহার করা যায়।)

2) কেন Lambda ব্যবহার করা হয় — typical use-cases

(a) sorted/list.sort-এ key দেয়ার জন্য

```
users = [('arman', 30), ('rakib', 25)]
users_sorted = sorted(users, key=lambda u: u[1])  # age অনুযায়ী sort
```

(b) map, filter

```
nums = [1,2,3,4]
squares = list(map(lambda x: x*x, nums))
evens = list(filter(lambda x: x%2==0, nums))
```

(c) reduce (functools)

```
from functools import reduce
prod = reduce(lambda a,b: a*b, [1,2,3,4])  # 24
```

(d) Inline callbacks (GUI, async, event handlers)

```
# hypothetical: btn.on_click(lambda e: print("Clicked", e))
```

(e) Quick small transforms in comprehensions or generator pipelines

```
funcs = [lambda x, i=i: x+i for i in range(3)]  # closure trick
[f(10) for f in funcs]  # [10,11,12]
```

3) Lambda vs def — কখন কোনটা?

- **Lambda ভালো:** এক-লাইনের তাত্ক্ষণিক, ছোট, পঠনযোগ্য, যখন ফাংশনটা ছোট ও নাম লাগবে না।
- **def ভালো:** জটিল লজিক, ডকস্ট্রিং দরকার, debugging/stack traces ম্যানেজ করা, annotations বা বহু লাইন হলে।
  নিয়ম-ভিত্তিক টিপ: যদি ফাংশনটি ১–২ লাইন সম্ভবত এবং খুব বার বার ব্যবহার হচ্ছে না → lambda ঠিক; অন্যথায় def ব্যবহার করো।

4) Lambda–এর সীমাবদ্ধতা (Important)

- কেবল একটি expression: statements (assignment, loops, try/except) লিখা যাবে না।
- ডিবাগ করা কঠিন: stack trace-এ <lambda> নামে দেখায় — পড়তে কম সুবিধাজনক।
- Pickling সমস্যা: সাধারণত lambda picklable নয় (serializing restrictions) — multiprocessing বা model persistence-এ সমস্যা হতে পারে।
- type hints/annotations প্রথাগতভাবে lambda–এ চেক বা attach করা কঠিন; def তে সুবিধা বেশি।

5) Closures & late binding — সতর্কতা (very important)

Lambda যখন লুপের মধ্যে closure তৈরি করে এবং সেই ভেরিয়েবল পরে বদলে যায়, সব lambda শেষের ভ্যালু দেখায় (late binding)। উদাহরণ:

```
funcs = [lambda: i for i in range(3)]
print([f() for f in funcs])  # [2,2,2]  <-- অধিকাংশকে বিস্মিত করে
```

সমাধান (common trick) — default argument capture:

```
funcs = [lambda i=i: i for i in range(3)]
print([f() for f in funcs])  # [0,1,2]
```

এখানে i=i প্রতিটি lambda-তে সেই সময়কার i-কে default হিসেবে সেভ করে।

6) Lambda as key in ML/data work (practical)

- Sorting columns, selecting top-k by score, grouping keys তৈরি ইত্যাদিতে lambda দরকারি:

```
rows = [{'id':1,'score':0.8}, {'id':2,'score':0.95}]
top = sorted(rows, key=lambda r: r['score'], reverse=True)
```

- সতর্কতা: অনেক ডেটা-heavy অপারেশনের জন্য Python-loop-based lambda ধীর হবে — vectorized ops (NumPy/Pandas) ব্যবহার করো:

BAD (loops in Python):

```
squared = list(map(lambda x: x*x, large_list))
```

BETTER (NumPy):

```
import numpy as np
arr = np.array(large_list)
squared = arr * arr  # vectorized, C-speed
```

7) Lambdas with multiple arguments and default values

```
adder = lambda x, y=10: x + y
print(adder(5))    # 15
print(adder(5,2))  # 7
```

---

8) Higher-order examples (factory pattern)

Lambda দিয়ে দ্রুত factory বানানো যায়:

```
def make_power(n):
    return lambda x: x**n

square = make_power(2)
cube = make_power(3)
print(square(4))  # 16
```

```
print(cube(2))   # 8
```

9) lambda + operator + functools — performance & readability

operator module-এর ফাংশনগুলো (যেমন operator.itemgetter, attrgetter, add) অনেক ক্ষেত্রেই lambda–র সরাসরি বিকল্প এবং দ্রুত:

```
from operator import itemgetter
rows = [('a', 3), ('b', 1)]
sorted(rows, key=itemgetter(1))  # faster & clearer than lambda r: r[1]
```

functools.partial-এর সাথে lambda কমই দরকার হয় কারণ partial দিয়ে অনেক inline customization করা যায়:

```
from functools import partial
def multiply(a,b): return a*b
double = partial(multiply, 2)  # equivalent to lambda x: multiply(2, x)
```

10) Lambdas এর ডিবাগিং / নামকরণ কৌশল

Lambda-কে একদম অপ্রচলিত জায়গায় ব্যবহার করলে ডিবাগ কঠিন। যদি প্রয়োজন হয় lambda-কে নাম দিতে পারো:

```
square = lambda x: x*x
print(square.__name__)  # '<lambda>'  (note: নাম descriptive নয়)
```

```
# Better: use def for readable name
def square_def(x):
   return x*x
```

functools.update_wrapper/wraps দিয়ে wrapper functions নাম ও docs কপি করা যায়, কিন্তু সাধারণত def করা সহজ।

11) Serialization / multiprocessing সমস্যা

- Lambdas সাধারণত picklable নয় (অর্থাৎ multiprocessing বা remote execution/worker process-এ পাঠাতে গেলে সমস্যা হতে পারে)। যদি multiprocessing দরকার হয়, ব্যবহার করো top-level def ফাংশন (module scope) — তখন pickle কাজ করবে।

12) Performance বিবেচনা

Lambda নিজে কোনও speed overhead রাখে না—এটা function object। কিন্তু Python-level looping/map/filter vs vectorized ops পারফরম্যান্সে ভিন্ন। ছোট কাজ-বিলাকের জন্য lambda ঠিক আছে; বড় সংখ্যা বা টেনসর-অপারেশনে NumPy/PyTorch/TensorFlow vectorized calls ব্যবহার করো।

13) AI/ML-স্পেসিফিক উদাহরণসমূহ

(a) custom key for sorting validation metrics

```
results = [{'name':'a','val':0.8},{'name':'b','val':0.85}]
best = max(results, key=lambda r: r['val'])
```

(b) transforming labels quickly

```
y = ['cat','dog','cat']
label_to_idx = {'cat':0, 'dog':1}
y_idx = list(map(lambda v: label_to_idx[v], y))
```

but pandas/map method or numpy is preferred for large data

(c) creating small feature transforms inline (but prefer Pipeline)

```
features = list(map(lambda row: [len(row['text'].split()), row['num']], rows))
```

For production use sklearn Pipeline / FunctionTransformer instead.

14) সর্বশেষ টিপস (best practices)

1. **Readable রাখো।** জটিল হলে def লিখো।
2. **Avoid heavy logic** inside lambda; keep them simple.
3. **Use operator.itemgetter / attrgetter** when possible for clarity & speed.
4. **Be careful with closures** — use i=i trick if capturing loop vars.
5. **Don't rely on pickling lambdas** — use top-level function definitions for multiprocessing/serialization.
6. **Prefer vectorized ops** (NumPy/Pandas) over map/lambda for numeric data.

---

15) ছোট অনুশীলনী (Practice — উত্তর নিচে দেওয়া আছে)

1. একটি lambda লিখো যা একটি সংখ্যার দ্বিগুণ ফেরত দেয়।
2. pairs = [(1,2),(3,0),(2,5)] sort করবে যাকে second element descending অনুযায়ী — lambda দিয়ে করো।
3. একটি make_multiplier(n) লিখে যেটা একটি lambda ফেরত দেয় যা ইনপুটকে n দিয়ে গুণ করে।
4. লুপে lambda তৈরি করে closure সমস্যা তৈরি করো এবং i=i fix দেখাও।
5. map ও filter দিয়ে একটি তালিকায় থেকে odd সংখ্যাগুলোর square তৈরি করো।

উত্তর (সংক্ষেপে):

1. double = lambda x: 2*x
2. sorted(pairs, key=lambda p: p[1], reverse=True)
3. def make_multiplier(n): return lambda x: x*n
4.

funcs = [lambda i=i: i for i in range(3)]

[f() for f in funcs] # [0,1,2]

5. list(map(lambda x: x*x, filter(lambda x: x%2==1, nums)))