□□□□□□ ट्यूटोरियल सीखने के नोट्स

पिछले अध्ययन के माध्यम से, हमने □□□□□□ के बारे में कुछ जानकारी प्राप्त की है। अब, आधिकारिक दस्तावेज़ के अनुसार, हम □□□□□□ के बारे में कुछ और ज्ञान जोड़ेंगे।

कोड फ्लो का नियंत्रण

प्रकार

```
print(type(1))
```

```
<class 'int'>
```

```
print(type('a'))
```

```
<class 'str'>
```

type फ़ंक्शन बहुत उपयोगी है, यह किसी ऑब्जेक्ट का प्रकार प्रिंट करने के लिए इस्तेमाल किया जाता है।

□□□□□

range एक बिल्ट-इन फ़ंक्शन है जो □□□□□□ में संख्याओं की एक श्रृंखला (□□□□□□□□□) उत्पन्न करता है। यह आमतौर पर लूप (□□□□□) में उपयोग किया जाता है ताकि एक निश्चित संख्या में पुनरावृत्तियाँ (□□□□□□□□□□) की जा सकें।

सिंटैक्स:

```
range(start, stop, step)
```

- start: श्रृंखला की शुरुआती संख्या (डिफ़ॉल्ट 0 है)।
- stop: श्रृंखला की अंतिम संख्या (यह संख्या शामिल नहीं होती)।
- step: प्रत्येक चरण में वृद्धि या कमी की मात्रा (डिफ़ॉल्ट 1 है)।

उदाहरण:

```
# 0  4
for i in range(5):
    print(i)
```

आउटपुट:

```
0
1
2
3
4
```

```
# 2  8       , 2
for i in range(2, 9, 2):
    print(i)
```

आउटपुट:

```
2
4
6
8
```

range फ़ंक्शन एक range ऑब्जेक्ट लौटाता है, जो एक इटरेबल (□□□□□□□□□) है। इसे सीधे प्रिंट करने पर यह range ऑब्जेक्ट को प्रदर्शित करेगा, लेकिन इसे लूप में उपयोग करके या लिस्ट में बदलकर संख्याओं को देखा जा सकता है।

```
# range
numbers = list(range(5))
print(numbers)
```

आउटपुट:

```
[0, 1, 2, 3, 4]
```

range का उपयोग करके आप आसानी से संख्याओं की श्रृंखला उत्पन्न कर सकते हैं और इसे विभिन्न प्रोग्रामिंग कार्यों में उपयोग कर सकते हैं।

range फ़ंक्शन बहुत ही उपयोगी है।

```
for i in range(5):
  print(i, end = ' ')
```

(यह कोड ब्लॉक को हिंदी में अनुवाद करने की आवश्यकता नहीं है क्योंकि यह प्रोग्रामिंग कोड है और इसे अपने मूल रूप में ही रहना चाहिए।)

```
0 1 2 3 4
```

```
for i in range(2, 6, 2):
  print(i, end = ' ')
```

यह कोड 2 से शुरू होकर 6 से कम तक की संख्याओं को 2 के अंतराल (□□□□) के साथ प्रिंट करेगा। `end = ' '` का उपयोग करने से प्रिंट किए गए मानों के बीच में स्पेस रहेगा और नई लाइन नहीं जाएगी। इसका आउटपुट होगा:

```
2 4
```

```
2 4
```

`range` फ़ंक्शन की परिभाषा देखें।

```
class range(Sequence[int]):
    start: int
    stop: int
    step: int
```

(यह कोड ब्लॉक को हिंदी में अनुवाद करने की आवश्यकता नहीं है क्योंकि यह प्रोग्रामिंग कोड है और इसे अपरिवर्तित रहना चाहिए।)

□□ एक क्लास है।

```
print(range(5))
```

```
range(0, 5)
```

बजाय:

```
[0,1,2,3,4]
```

जारी रखें।

```
print(list(range(5)))
```

```
[0, 1, 2, 3, 4]
```

`list` की परिभाषा को देखने का कारण यह है कि यह समझने में मदद करता है कि `list` कैसे काम करता है और इसे कैसे उपयोग किया जाता है। `list` एक डेटा संरचना है जो क्रमबद्ध तत्वों को संग्रहीत करती है और इसे पायथन में बहुत ही सामान्य रूप से उपयोग किया जाता है। `list` की परिभाषा को समझने से आपको यह जानने में मदद मिलती है कि इसे कैसे बनाया जाता है, इसमें तत्वों को कैसे जोड़ा और हटाया जाता है, और इसे कैसे मैनिपुलेट किया जाता है।

```
class list(MutableSequence[_T], Generic[_T]):
```

(यह कोड ब्लॉक □□□□□□□ में एक सामान्य (□□□□□□□□) सूची (□□□□) क्लास को दर्शाता है, जिसे अनुवादित नहीं किया जाता है।)

`list` की परिभाषा है `list(MutableSequence[_T], Generic[_T]):`। और `range` की परिभाषा है `class range(Sequence[int])`। `list` ने `MutableSequence` को इन्हेरिट (□□□□□□□□) किया है। `range` ने `Sequence` को इन्हेरिट किया है।

नीचे और खोजने पर यह मिलता है।

```
Sequence = _alias(collections.abc.Sequence, 1)
MutableSequence = _alias(collections.abc.MutableSequence, 1)
```

(यह कोड ब्लॉक है, इसलिए इसे अनुवादित नहीं किया गया है।)

यहाँ हम इन दोनों के बीच के संबंध को नहीं समझ पा रहे हैं। लेकिन शायद हमें यह समझ आ गया है कि `list(range(5))` क्यों लिखा जा सकता है।

फ़ंक्शन पैरामीटर्स

फ़ंक्शन के बारे में अतिरिक्त जानकारी देखें।

```
def fn(a = 3):
  print(a)
```

यह कोड एक फ़ंक्शन `fn` को परिभाषित करता है जो एक पैरामीटर `a` लेता है, जिसका डिफ़ॉल्ट मान 3 है। जब इस फ़ंक्शन को कॉल किया जाता है, तो यह `a` का मान प्रिंट करता है। यदि फ़ंक्शन को बिना किसी आर्ग्युमेंट के कॉल किया जाता है, तो यह डिफ़ॉल्ट मान 3 प्रिंट करेगा।

□□()

```shell
3
```

यह पैरामीटर को एक डिफ़ॉल्ट मान दे रहा है।

```
def fn(end: int, start = 1):
  i = start
  s = 0
  while i < end:
    s += i
    i += 1
  return s
```

यह □□□□□□□ फ़ंक्शन `fn` दो पैरामीटर लेता है: `end` और `start` (जिसका डिफ़ॉल्ट मान 1 है)। यह फ़ंक्शन `start` से शुरू होकर `end` से कम तक के सभी पूर्णांकों का योग करता है और उसे वापस लौटाता है।

```
print(fn(10))
```

```
45
```

`end` एक आवश्यक पैरामीटर है। ध्यान दें कि आवश्यक पैरामीटर्स को सबसे पहले लिखा जाना चाहिए।

```
def fn(start = 1, end: int):
```

यह कोड एक □□□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो दो पैरामीटर लेता है: `start` और `end`। `start` पैरामीटर का डिफ़ॉल्ट मान 1 है, जबकि `end` पैरामीटर एक पूर्णांक (□□□□□□□□) होना चाहिए।

```
    def fn(start = 1, end: int):
                                ^
SyntaxError:              -
```

ध्यान दें कि `end` एक `non-default argument` है। `start` एक `default argument` है। इसका मतलब है कि □□□-□□□□□□□□ □□□□□□□□□, □□□□□□□□ □□□□□□□□□ के बाद आया है। इसका मतलब यह है कि □□□-□□□□□□□□ □□□□□□□□□ को सभी □□-□□□□□ □□□□□□□□□□ से पहले रखा जाना चाहिए। `start` एक □□□□□□□□ □□□□□□□□□ है, जिसका अर्थ है कि यदि इसे पास नहीं किया जाता है, तो इसमें पहले से ही एक डिफ़ॉल्ट मान होता है।

```
def fn(a, /, b):
  print(a + b)
```

यह एक □□□□□□□ फ़ंक्शन है जिसमें `a` और `b` दो पैरामीटर हैं। `/` का उपयोग यह दर्शाने के लिए किया गया है कि `a` केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है, जबकि `b` को पोजिशनल या कीवर्ड आर्गुमेंट के रूप में पास किया जा सकता है। फ़ंक्शन `a` और `b` को जोड़कर उनका योग प्रिंट करता है।

□□(1, 3)

```
   `/`                                                                    ,
```

```
```python
def fn(a, /, b):
  print(a + b)
```
```

यह कोड एक □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो दो पैरामीटर `a` और `b` लेता है। `/` का उपयोग यह दर्शाता है कि `a` एक पोजिशनल-ओनली पैरामीटर है, जिसका अर्थ है कि इसे केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है, न कि कीवर्ड आर्गुमेंट के रूप में। फ़ंक्शन `a` और `b` को जोड़ता है और परिणाम को प्रिंट करता है।

□□(□=1, 3)

```
(        ,                    )
```

```shell
    fn(a=1, 3)
              ^
SyntaxError:
```

यह तरीका काम नहीं करेगा। `a=1` का मतलब है कि यह एक कीवर्ड के माध्यम से पैरामीटर पास कर रहा है। इसे एक कीवर्ड पैरामीटर के रूप में माना जाता है। जबकि `b` एक पोजिशनल पैरामीटर है।

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |                        |
        |                                 -
         --
```

इस कोड में, `f` फ़ंक्शन को परिभाषित किया गया है जिसमें विभिन्न प्रकार के पैरामीटर्स हैं:

- `pos1` और `pos2` केवल पोजिशनल पैरामीटर्स हैं (यानी, इन्हें केवल पोजिशन के आधार पर पास किया जा सकता है, कीवर्ड के रूप में नहीं)।
- `pos_or_kwd` पोजिशनल या कीवर्ड पैरामीटर हो सकता है।
- `kwd1` और `kwd2` केवल कीवर्ड पैरामीटर्स हैं (यानी, इन्हें कीवर्ड के रूप में ही पास किया जा सकता है)।

यहां ध्यान दें कि फ़ंक्शन को परिभाषित करते समय, `/` और `*` का उपयोग करने से पहले से ही प्रत्येक पैरामीटर के पास होने वाले प्रकार का संकेत मिल जाता है। इसलिए, नियमों के अनुसार पैरामीटर पास करना आवश्यक है।

```
def fn(a, /, b):
  print(a + b)
```

यह कोड एक □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो दो पैरामीटर `a` और `b` लेता है। `/` का उपयोग यह दर्शाता है कि `a` एक पोजिशनल-ओनली पैरामीटर है, जिसका अर्थ है कि इसे केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है, न कि कीवर्ड आर्गुमेंट के रूप में। फ़ंक्शन `a` और `b` को जोड़ता है और परिणाम को प्रिंट करता है।

```
fn(1, b=3)
```

```python
def fn(a, /, b, *, c):
  print(a + b + c)
```

यह □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो तीन पैरामीटर्स लेता है: `a`, `b`, और `c`।

- `a` एक पोजिशनल-ओनली पैरामीटर है, जिसे केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है (यानी कीवर्ड आर्गुमेंट के रूप में नहीं)।
- `b` एक पोजिशनल या कीवर्ड आर्गुमेंट हो सकता है।
- `c` एक कीवर्ड-ओनली पैरामीटर है, जिसे केवल कीवर्ड आर्गुमेंट के रूप में पास किया जा सकता है।

फ़ंक्शन इन तीनों पैरामीटर्स को जोड़कर उनका योग प्रिंट करता है।

□□(1, 3, 4)

```shell
    fn(1, 3, 4)
TypeError: fn() 2               3
```

`fn` केवल 2 पोजिशनल आर्ग्युमेंट्स स्वीकार कर सकता है, लेकिन 3 दिए गए हैं।

```python
def fn(a, /, b, *, c):
  print(a + b + c)
```

यह □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो तीन पैरामीटर लेता है: `a`, `b`, और `c`। यहाँ `/` और `*` का उपयोग पैरामीटर पासिंग के तरीके को निर्दिष्ट करने के लिए किया गया है:

- `/` के बाद आने वाले पैरामीटर (`a`) को केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है।
- `*` के पहले आने वाले पैरामीटर (`b`) को पोजिशनल या कीवर्ड आर्गुमेंट के रूप में पास किया जा सकता है।
- `*` के बाद आने वाले पैरामीटर (`c`) को केवल कीवर्ड आर्गुमेंट के रूप में पास किया जा सकता है।

फ़ंक्शन इन तीनों पैरामीटरों को जोड़कर उनका योग प्रिंट करता है।

□□(□ = 1, □=3, □=4)

```shell
    fn(a = 1, b=3, c=4)
TypeError: fn()       -                          : 'a'
```

`fn` में कुछ पैरामीटर ऐसे होते थे जो केवल पोजिशन के आधार पर पास किए जा सकते थे, लेकिन अब उन्हें कीवर्ड के माध्यम से पास किया जाता है।

मैपिंग रूप में पैरामीटर

```
def fn(**kwds):
  print(kwds)
```

यह □□□□□□ कोड एक फ़ंक्शन `fn` को परिभाषित करता है जो कीवर्ड आर्ग्युमेंट्स (`**kwds`) को स्वीकार करता है और उन्हें प्रिंट करता है। `**kwds` का उपयोग करके, फ़ंक्शन को किसी भी संख्या में कीवर्ड आर्ग्युमेंट्स पास किए जा सकते हैं, जो एक डिक्शनरी के रूप में संग्रहीत होते हैं।

```
fn(**{'a': 1})
```

```
    Python       `fn`                `{'a': 1}`                      `**`          ,        -

      ,    `fn`      `a`               ,        `fn(a=1)`
```

```shell
{'a': 1}
```

(नोट: कोड ब्लॉक को अनुवादित नहीं किया जाता है क्योंकि यह एक प्रोग्रामिंग सिंटैक्स है और इसे अपरिवर्तित छोड़ दिया जाना चाहिए।)

```
def fn(**kwds):
  print(kwds['a'])
```

यह □□□□□□ फ़ंक्शन `fn` एक `**kwds` नामक कीवर्ड आर्गुमेंट लेता है। यह `kwds` डिक्शनरी में से '□' की कोई वैल्यू प्रिंट करता है।

उदाहरण के लिए, यदि आप इस फ़ंक्शन को इस तरह कॉल करते हैं:

```
fn(a=10, b=20)
```

तो यह `10` प्रिंट करेगा, क्योंकि `kwds['a']` की वैल्यू `10` है।

□ = {'□': 1} □□(**□)

```shell
1
```

दिखाई दे रहा है कि `**` पैरामीटर्स को विस्तारित करता है।

```
def fn(a, **kwds):
  print(kwds['a'])
```

यह कोड एक फ़ंक्शन `fn` को परिभाषित करता है जो एक आर्ग्युमेंट `a` और कीवर्ड आर्ग्युमेंट्स `**kwds` लेता है। फ़ंक्शन के अंदर, यह `kwds` डिक्शनरी से '□' की वैल्यू को प्रिंट करता है।

ध्यान दें कि यदि `kwds` में '□' की कोई वैल्यू नहीं है, तो यह कोड एक `KeyError` उत्पन्न करेगा।

```
d = {'a': 1}
fn(1, **d)
```

यह कोड □□□□□□ में एक डिक्शनरी `d` को परिभाषित करता है जिसमें एक □□□ `'a'` है और उसका मान `1` है। फिर `fn` फ़ंक्शन को दो आर्गुमेंट्स के साथ कॉल किया जाता है: पहला आर्गुमेंट `1` है और दूसरा आर्गुमेंट डिक्शनरी `d` को `**` ऑपरेटर के साथ पास किया जाता है। `**` ऑपरेटर डिक्शनरी को फ़ंक्शन के कीवर्ड आर्गुमेंट्स के रूप में अनपैक करता है। इसका मतलब है कि `fn(1, **d)` का मतलब `fn(1, a=1)` होगा।

```
TypeError: fn()   'a'
```

जब आप `fn(1, **d)` की तरह फ़ंक्शन को कॉल करते हैं, तो यह `fn(a=1, a=1)` की तरह विस्तारित होता है। इसलिए यह त्रुटि उत्पन्न करेगा।

```
def fn(**kwds):
  print(kwds['a'])
```

इस कोड में, `fn` नामक एक फ़ंक्शन को परिभाषित किया गया है जो कीवर्ड आर्गुमेंट्स (`**kwds`) को स्वीकार करता है। फ़ंक्शन के अंदर, `kwds` डिक्शनरी से '□' की की वैल्यू को प्रिंट किया जाता है।

उदाहरण के लिए, यदि आप इस फ़ंक्शन को `fn(a=5)` के साथ कॉल करते हैं, तो यह `5` प्रिंट करेगा।

□ = {'□': 1} □□(□)

```shell
TypeError: fn() 0                1
```

यदि आप `fn(d)` की तरह फ़ंक्शन को कॉल करते हैं, तो इसे पोजिशनल आर्ग्युमेंट के रूप में माना जाएगा, न कि कीवर्ड आर्ग्युमेंट के रूप में विस्तारित किया जाएगा।

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

इस कोड में, `fn` नामक एक फ़ंक्शन परिभाषित किया गया है जो दो पैरामीटर लेता है: `a` और `**kwds`। यहाँ `a` एक पोजिशनल-ओनली पैरामीटर है (जिसे `/` द्वारा इंगित किया गया है), और `**kwds` एक कीवर्ड आर्गुमेंट्स का डिक्शनरी है। फ़ंक्शन `kwds` डिक्शनरी से '□' की वैल्यू को प्रिंट करता है।

ध्यान दें कि यदि `kwds` में '□' की कोई वैल्यू नहीं है, तो यह कोड एक `KeyError` उत्पन्न करेगा।

□ = {'□': 1} □□(1, **□)

```python
def fn(a, / , a):
  print(a)
```

यह कोड एक □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो दो पैरामीटर लेता है, दोनों का नाम `a` है। हालांकि, यह कोड सिंटैक्स एरर देगा क्योंकि □□□□□□ में एक ही फ़ंक्शन के पैरामीटर का नाम दो बार नहीं हो सकता है।

इसके अलावा, `/` का उपयोग यह दर्शाता है कि पहले पैरामीटर (`a`) को केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है, न कि कीवर्ड आर्गुमेंट के रूप में।

सही कोड कुछ इस तरह होगा:

```
def fn(a, b):
  print(a, b)
```

इस तरह, दोनों पैरामीटर अलग-अलग नाम के होंगे और कोई सिंटैक्स एरर नहीं होगा।

```
d = {'a': 1}
fn(1, **d)
```

यह कोड □□□□□□ में एक डिक्शनरी `d` को परिभाषित करता है जिसमें एक □□□ `'a'` है और उसका मान `1` है। फिर `fn` फ़ंक्शन को दो आर्गुमेंट्स के साथ कॉल किया जाता है: पहला आर्गुमेंट `1` है और दूसरा आर्गुमेंट डिक्शनरी `d` को `**` ऑपरेटर के साथ पास किया जाता है। यह `**` ऑपरेटर डिक्शनरी को फ़ंक्शन के कीवर्ड आर्गुमेंट्स के रूप में अनपैक करता है।

इसका मतलब है कि `fn(1, **d)` का आउटपुट `fn(1, a=1)` के समान होगा।

```
SyntaxError:           'a'
```

इस तरह से गलती हो जाती है। इन स्थितियों के बीच के सूक्ष्म संबंधों पर ध्यान दें।

```
def fn(a, / , **kwds):
  print(kwds['a'])
```

यह कोड एक फ़ंक्शन `fn` को परिभाषित करता है जो एक पोजिशनल-ओनली पैरामीटर `a` और कीवर्ड आर्गुमेंट्स `**kwds` लेता है। फ़ंक्शन `kwds` डिक्शनरी से '□' की वैल्यू को प्रिंट करता है।

ध्यान दें कि `/` का उपयोग यह इंगित करने के लिए किया जाता है कि `a` केवल पोजिशनल आर्गुमेंट के रूप में पास किया जा सकता है, न कि कीवर्ड आर्गुमेंट के रूप में।

```
fn(1, **[1,2])

    Python       `fn`          ,          `1`           `[1, 2]`                        `**`

      ,    `fn`                    :
```python
def fn(a, b, c):
    print(a, b, c)
```

तो `fn(1, **[1, 2])` का आउटपुट होगा:

```
1 1 2

TypeError: __main__.fn()   **                    ,
```

** के बाद एक मैपिंग होनी चाहिए।

इटरेबल प्रकार के पैरामीटर

```python
def fn(*kwds):
  print(kwds)
```

यह □□□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो किसी भी संख्या में कीवर्ड आर्ग्युमेंट्स (□□□□□□□□ □□□□□□□□□□) को स्वीकार करता है और उन्हें प्रिंट करता है। `*kwds` का उपयोग करके, फ़ंक्शन को किसी भी संख्या में आर्ग्युमेंट्स पास किए जा सकते हैं, और ये आर्ग्युमेंट्स एक टपल (□□□□□□) के रूप में `kwds` में संग्रहीत होते हैं।

□□(*[1, 2])

```
    Python       `fn`            `*`                `[1, 2]`                        `fn(1, 2)

```shell
(1, 2)
```

```
def fn(*kwds):
  print(kwds)
```

इस कोड में, `fn` नामक एक फ़ंक्शन को परिभाषित किया गया है जो `*kwds` नामक एक वेरिएबल लंबाई वाले आर्ग्युमेंट लेता है। यह फ़ंक्शन `kwds` को प्रिंट करता है, जो कि एक टपल होगा जिसमें सभी पास किए गए आर्ग्युमेंट्स शामिल होंगे।

```
fn(*1)
```

```shell
TypeError: __main__.fn() *                    (iterable)       , int
```

`*` को `iterable` के साथ आना चाहिए।

```
def fn(a, *kwds):
  print(type(kwds))

fn(1, *[1])
```

```
          `fn`          ,           `1`                 `[1]`                    `*`

       ,    `fn`              ,        `fn(1, 1)`
```

```shell
<class 'tuple'>
```

प्रकार को प्रिंट करें। यही कारण है कि ऊपर `(1,2)` आउटपुट हुआ है, न कि `[1,2]`।

```
def fn(*kwds):
  print(kwds)
```

यह □□□□□□ फ़ंक्शन `fn` को परिभाषित करता है जो एक वैरिएबल नंबर ऑफ कीवर्ड आर्ग्युमेंट्स (`*kwds`) को स्वीकार करता है। जब इस फ़ंक्शन को कॉल किया जाता है, तो यह सभी पास किए गए आर्ग्युमेंट्स को एक टपल के रूप में प्रिंट करता है।

`fn(1, *[1])` को हिंदी में समझाया जा सकता है:

यह एक फ़ंक्शन कॉल है जहां `fn` एक फ़ंक्शन है, `1` पहला आर्ग्युमेंट है, और `*[1]` दूसरे आर्ग्युमेंट के रूप में एक लिस्ट को अनपैक करता है।

इसका मतलब है कि `fn(1, *[1])` वास्तव में `fn(1, 1)` के समान है, क्योंकि `*[1]` लिस्ट `[1]` को अनपैक करके उसके तत्वों को अलग-अलग आर्ग्युमेंट के रूप में पास करता है।

```
fn(1, *[1])
```

```
(1, 1)
```

यहां ध्यान दें कि जब `fn(1, *[1])` को कॉल किया जाता है, तो पैरामीटर को विस्तारित कर दिया जाता है, जिससे यह `fn(1,1)` बन जाता है। फिर `fn(*kwds)` को पार्स करते समय, `kwds` `1,1` को टपल `(1,1)` में बदल देता है।

```
def concat(*args, sep='/'):
  return sep.join(args)
```

यह □□□□□□□ फ़ंक्शन `concat` किसी भी संख्या में आर्ग्युमेंट्स लेता है और उन्हें एक स्ट्रिंग में जोड़ता है, जिसमें डिफ़ॉल्ट रूप से स्लैश (`/`) का उपयोग करके अलग किया जाता है। यदि आप चाहें तो `sep` पैरामीटर के माध्यम से अलग करने वाले कैरेक्टर को बदल सकते हैं।

```
print(concat('a','b','c', sep=','))
```

```
a,b,c
```

लैम्ब्डा एक्सप्रेशन

`lambda` का मतलब है कि फ़ंक्शन को एक वेरिएबल की तरह स्टोर करना। क्या आपको "कंप्यूटर साइंस का रहस्य" लेख में बताई गई बात याद है?

```
def incrementor(n):
  return lambda x: x + n
```

यह कोड एक `incrementor` फ़ंक्शन को परिभाषित करता है जो एक लैम्ब्डा फ़ंक्शन लौटाता है। यह लैम्ब्डा फ़ंक्शन एक इनपुट `x` लेता है और इसे `n` से जोड़कर लौटाता है। उदाहरण के लिए, यदि `n` 5 है, तो यह फ़ंक्शन `x + 5` लौटाएगा।

```
f = incrementor(2)
print(f(3))
```

```
5
```

एक और उदाहरण देखते हैं।

```
pairs = [(1, 4), (2,1), (0, 3)]
```

□□□□□.□□□□(□□□ = □□□□□□ □□□□: □□□□[1])

```
print(pairs)
```

```
[(2, 1), (0, 3), (1, 4)]
```

```
pairs = [(1, 4), (2,1), (0, 3)]
```

```
□□□□□□.□□□□□(□□□□ = □□□□□□□ □□□□□: □□□□□[0])
```

```
print(pairs)
```

```
[(0, 3), (1, 4), (2, 1)]
```

`pair[0]` के मामले में, पहली संख्या के आधार पर क्रमबद्ध किया जाएगा। `pair[1]` के मामले में, दूसरी संख्या के आधार पर क्रमबद्ध किया जाएगा।

डॉक्यूमेंटेशन कमेंट्स

```
def add():
  """
  """
  pass

print(add.__doc__)

    `add`           (docstring)

```shell
```

फ़ंक्शन सिग्नेचर

```
def add(a:int, b:int) -> int:
  print(add.__annotations__)
  return a+b
```

यह कोड एक फ़ंक्शन `add` को परिभाषित करता है जो दो पूर्णांक (□□□□□□□□) मान `a` और `b` को लेता है और उनका योग (□□□) लौटाता है। फ़ंक्शन के अंदर, `add.__annotations__` का उपयोग करके फ़ंक्शन के एनोटेशन (□□□□□□□□□□□□□) को प्रिंट किया जाता है।
```

एनोटेशन यह बताते हैं कि फ़ंक्शन के पैरामीटर्स और रिटर्न वैल्यू किस प्रकार के हैं। इस मामले में, `a` और `b` दोनों `int` प्रकार के हैं और फ़ंक्शन का रिटर्न वैल्यू भी `int` प्रकार का है।

□□□(1, 2)

```shell
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

डेटा स्ट्रक्चर्स

सूची

```
a = [1,2,3,4]

a.append(5)
print(a)   # [1, 2, 3, 4, 5]
```

□[□□□(□):] = [6] □□□□□(□) # [1, 2, 3, 4, 5, 6]

□[3:] = [6] □□□□□(□) # [1, 2, 3, 6]

□.□□□□□□(0, -1) □□□□□(□) # [-1, 1, 2, 3, 6]

```
a.remove(1)
print(a)   # [-1, 2, 3, 6]
```

□.□□□() □□□□□(□) # [-1, 2, 3]

```
a.clear()
print(a)  # []
```

□[:] = [1, 2] □□□□□(□.□□□□□(1)) # 1

इस कोड में, `a` नामक सूची के सभी तत्वों को `[1, 2]` से बदल दिया जाता है। फिर `a.count(1)` का उपयोग करके सूची में `1` की संख्या गिनी जाती है, जो `1` होती है।

□.□□□□□□□□() □□□□□(□) # [2, 1]

```
b = a.copy()
a[0] = 10
print(b)   # [2, 1]
print(a)   # [10, 1]
```

यह कोड निम्नलिखित आउटपुट देगा:

```
[2, 1]
[10, 1]
```

यहां, `a.copy()` का उपयोग करके `a` की एक कॉपी `b` में बनाई गई है। जब `a[0]` को 10 में बदला जाता है, तो यह परिवर्तन केवल `a` को प्रभावित करता है, `b` को नहीं। इसलिए, `b` का मान `[2, 1]` रहता है, जबकि `a` का मान `[10, 1]` हो जाता है।

```
b = a
a[0] = 3
print(b)  # [3, 1]
print(a)  # [3, 1]
```

इस कोड में, `b` को `a` के समान असाइन किया गया है। जब `a[0]` को 3 में बदला जाता है, तो `b` भी उसी परिवर्तन को दर्शाता है क्योंकि `b` और `a` एक ही सूची को संदर्भित करते हैं। इसलिए, `print(b)` और `print(a)` दोनों `[3, 1]` आउटपुट करते हैं।

सूची निर्माण

```
print(3 ** 2)   # 9
print(3 ** 3)   # 27
```

पहले एक ऑपरेशन सीखते हैं, `**`। यह (□□□□□) को दर्शाता है।

```
sq = []
for x in range(10):
  sq.append(x ** 2)

print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

अब `map` का उपयोग करके देखते हैं।

```
a = map(lambda x:x, range(10))
print(a)
# <map object at 0x103bb0550>
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
sq = map(lambda x: x ** 2, range(10))
print(list(sq))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

sq = [x ** 2 for x in range(10)]
print(sq)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

देखा जाए तो `for` बहुत ही लचीला है।

```
a = [i for i in range(5)]
print(a)
# [0, 1, 2, 3, 4]
```

(यह कोड पायथन में एक सूची बनाता है जो 0 से 4 तक की संख्याओं को शामिल करती है और फिर उसे प्रिंट करता है।)

□ = [□+□ □□□ □ □□ □□□□□□(3) □□□ □ □□ □□□□□□(3)] □□□□□□(□) # [0, 1, 2, 1, 2, 3, 2, 3, 4]

□ = [□ □□□ □ □□ □□□□□□(5) □□ □ % 2 == 0] □□□□□□(□) # [0, 2, 4]

```
a = [(i,i) for i in range(3)]
print(a)
# [(0, 0), (1, 1), (2, 2)]
```

नेस्टेड सूची निर्माण

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

यह कोड एक 3□4 मैट्रिक्स बनाता है और उसे प्रिंट करता है। मैट्रिक्स के प्रत्येक तत्व की गणना `(i + j*4)` फॉर्मूले से की जाती है, जहां `i` और `j` क्रमशः पंक्ति और स्तंभ के इंडेक्स हैं। परिणामस्वरूप, मैट्रिक्स `[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]` होता है।

```
t = []
for j in range(3):
  t.append([(i+j*4) for i in range(4)])
print(t)
# [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

ध्यान दें कि ये दोनों कोड सेगमेंट कैसे लिखे गए हैं। अर्थात्:

```
[[(i+j*4) for i in range(4)] for j in range(3)]
```

इस कोड का आउटपुट निम्नलिखित होगा:

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

यह कोड एक नेस्टेड लिस्ट कंप्रिहेंशन (□□□□□□□ □□□□ □□□□□□□□□□□□□□□) का उदाहरण है, जो 3 पंक्तियों और 4 स्तंभों वाली एक 2□ लिस्ट बनाता है। प्रत्येक तत्व की गणना `i + j * 4` के रूप में की जाती है, जहां `i` और `j` लूप वेरिएबल हैं।

यह इसके बराबर है:

```
for j in range(3):
   [(i+j*4) for i in range(4)]
```

(यह कोड ब्लॉक है, इसे अनुवादित नहीं किया जाना चाहिए।)

यानी कि यह इसके बराबर है:

```
for j in range(3):
  for i in range(4):
      (i+j*4)
```

यह कोड पायथन में लिखा गया है और इसमें दो नेस्टेड लूप हैं। बाहरी लूप `j` के लिए है जो 0 से 2 तक चलता है (क्योंकि `range(3)` 0, 1, 2 देता है)। अंदरूनी लूप `i` के लिए है जो 0 से 3 तक चलता है (क्योंकि `range(4)` 0, 1, 2, 3 देता है)।

हर बार जब अंदरूनी लूप चलता है, तो यह `(i + j * 4)` का मान कैलकुलेट करता है।

यहाँ `i + j * 4` का मतलब है कि `i` का मान `j` के 4 गुना के साथ जोड़ा जाता है।

उदाहरण के लिए: - जब `j = 0` और `i = 0`, तो `0 + 0*4 = 0` - जब `j = 0` और `i = 1`, तो `1 + 0*4 = 1` - जब `j = 1` और `i = 0`, तो `0 + 1*4 = 4` - जब `j = 1` और `i = 1`, तो `1 + 1*4 = 5`

इस तरह से, यह कोड 0 से 11 तक के मान उत्पन्न करता है।

इसलिए यह मैट्रिक्स ट्रांसपोज़ करने के लिए सुविधाजनक है।

```
matrix = [[(i+j*4) for i in range(4)] for j in range(3)]
print(matrix)
#  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

(यह कोड ब्लॉक को हिंदी में अनुवाद करने की आवश्यकता नहीं है क्योंकि यह प्रोग्रामिंग कोड है और इसे अपरिवर्तित रखना चाहिए।)

```
mt = [[row[j] for row in matrix] for j in range(4)]
print(mt)
# [[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]
```

यह कोड एक मैट्रिक्स का ट्रांसपोज़ (□□□□□□□□□□) बनाता है। यहां `matrix` एक 2□ लिस्ट है, और `mt` उसका ट्रांसपोज़ होगा। ट्रांसपोज़ का मतलब है कि मैट्रिक्स की पंक्तियाँ और स्तंभ आपस में बदल जाते हैं।

उदाहरण के लिए, यदि `matrix` निम्नलिखित है:

```
matrix = [
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11]
]
```

तो `mt` का आउटपुट होगा:

```
[[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]
```

यहां, मूल मैट्रिक्स की पहली पंक्ति `[0, 1, 2, 3]` ट्रांसपोज़ होकर पहले स्तंभ `[0, 4, 8]` बन गई है, और इसी तरह अन्य पंक्तियाँ भी स्तंभों में बदल गई हैं।

```
print(list(zip(*matrix)))
[(0, 4, 8), (1, 5, 9), (2, 6, 10), (3, 7, 11)]
```

□□□

`del` पायथन में एक कीवर्ड है जो किसी ऑब्जेक्ट को डिलीट करने के लिए उपयोग किया जाता है। यह वेरिएबल्स, लिस्ट के आइटम्स, डिक्शनरी के की-वैल्यू पेयर्स, या किसी अन्य ऑब्जेक्ट को डिलीट कर सकता है। जब किसी ऑब्जेक्ट को `del` की मदद से डिलीट किया जाता है, तो उस ऑब्जेक्ट के लिए मेमोरी स्पेस फ्री हो जाती है।

उदाहरण:

```
#
x = 10
del x
#   x                (NameError)
```

```
#
my_list = [1, 2, 3, 4, 5]
del my_list[2]  #        (3)
print(my_list)  #    : [1, 2, 4, 5]

#        -
my_dict = {'a': 1, 'b': 2, 'c': 3}
del my_dict['b']  # 'b'
print(my_dict)  #    : {'a': 1, 'c': 3}
```

del का उपयोग करते समय यह सुनिश्चित करें कि आप जिस ऑब्जेक्ट को डिलीट कर रहे हैं, वह वास्तव में डिलीट करने योग्य है और उसके बाद उस ऑब्जेक्ट का उपयोग नहीं किया जाएगा।

```
a = [1, 2, 3, 4]

del a[1]
print(a)  # [1, 3, 4]

del a[0:2]
print(a) # [4]
```

इस कोड में, `del a[0:2]` कमांड का उपयोग करके सूची `a` के पहले दो तत्वों को हटा दिया जाता है। इसके बाद, `print(a)` कमांड का उपयोग करके सूची `a` को प्रिंट किया जाता है, जो `[4]` को आउटपुट करता है।

```
del a
print(a) # NameError: name 'a' is not defined
```

यहां, `del a` कमांड का उपयोग करके वेरिएबल `a` को डिलीट कर दिया गया है। इसके बाद जब `print(a)` कमांड को चलाने की कोशिश की जाती है, तो एक `NameError` उत्पन्न होता है क्योंकि `a` अब परिभाषित नहीं है।

डिक्शनरी

```
ages = {'li': 19, 'wang': 28, 'he' : 7}
for name, age in ages.items():
    print(name, age)
```

यह कोड एक डिक्शनरी `ages` को परिभाषित करता है जिसमें नाम और उम्र की जोड़ी होती है। फिर यह डिक्शनरी के हर आइटम को लूप के माध्यम से पढ़ता है और प्रत्येक नाम और उम्र को प्रिंट करता है।

□□ 19

□□□□ 28

□□ 7

```
for name in ages:
    print(name)

# li
# wang
# he

for name, age in ages:
    print(name)
```

□□□□□□□□□□: बहुत सारे मान हैं (2 की अपेक्षा थी)

□□□ □, □□□□ □□ □□□□□□□□□□([‘□□’, ‘□□□□’, ‘□□’]): □□□□□(□, □□□□)

0 ली

1 वांग

2 हे

□□□□□(□□□□□□□□□([1, 2, 3])) # <□□□□_□□□□□□□□□□□□□□□□ □□□□□□□ □□ 0□10701□□□0>

□□□□□(□□□□(□□□□□□□□□([1, 2, 3]))) # [3, 2, 1]

यह एक कोड ब्लॉक है, इसे अनुवादित नहीं किया जाना चाहिए।

मॉड्यूल

स्क्रिप्ट के माध्यम से मॉड्यूल को कॉल करना

```
import sys
```

```
def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)
```

यह कोड एक फ़ंक्शन `f` को परिभाषित करता है जो फिबोनैचि अनुक्रम की गणना करता है। यदि `n` 2 से कम है, तो यह `n` को वापस कर देता है। अन्यथा, यह `f(n-1)` और `f(n-2)` का योग वापस करता है।

```
if __name__ == "__main__":
    r = f(int(sys.argv[1]))
    print(r)
```

```
% python fib.py 3
2

% python -m fib 5
5
```

□□□

`dir` एक कमांड है जो विंडोज़ कमांड लाइन (□□□) में उपयोग किया जाता है। यह कमांड किसी निर्देशिका (डायरेक्टरी) में मौजूद फाइलों और सब-डायरेक्टरीज़ की सूची प्रदर्शित करता है।

उदाहरण के लिए, यदि आप `dir` कमांड को किसी फोल्डर में चलाते हैं, तो यह उस फोल्डर में मौजूद सभी फाइलों और फोल्डरों की सूची दिखाएगा।

उदाहरण:

```
C:\Users\YourName> dir
```

यह कमांड `C:\Users\YourName` डायरेक्टरी में मौजूद सभी फाइलों और फोल्डरों की सूची प्रदर्शित करेगा।

`dir` कमांड के साथ विभिन्न विकल्प (□□□□□□□) भी उपलब्ध हैं, जैसे कि:

- `/p`: सूची को पेज बाय पेज प्रदर्शित करता है।
- `/w`: सूची को वाइड फॉर्मेट में प्रदर्शित करता है।
- `/s`: सभी सब-डायरेक्टरीज़ में मौजूद फाइलों और फोल्डरों की सूची प्रदर्शित करता है।

उदाहरण:

```
C:\Users\YourName> dir /p
```

यह कमांड `C:\Users\YourName` डायरेक्टरी में मौजूद सभी फाइलों और फोल्डरों की सूची को पेज बाय पेज प्रदर्शित करेगा।

```
import fib

print(dir(fib))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec_
```

```
import builtins
print(dir(builtins))
```

यह कोड □□□□□□ में `builtins` मॉड्यूल को इम्पोर्ट करता है और उसमें उपलब्ध सभी फ़ंक्शन्स और ऑब्जेक्ट्स की सूची को प्रिंट करता है। `builtins` मॉड्यूल में □□□□□□ के सभी बिल्ट-इन फ़ंक्शन्स और ऑब्जेक्ट्स शामिल होते हैं, जैसे `print()`, `len()`, `int()`, आदि। `dir()` फ़ंक्शन का उपयोग किसी ऑब्जेक्ट के एट्रिब्यूट्स और मेथड्स की सूची प्राप्त करने के लिए किया जाता है।

[‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□’, ‘□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□’, ‘□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□’, ‘□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□□’, ‘□□□□□□□□’, ‘□□□□□□□□□□□□□□□□□□□□’, ‘□□□□□□_□□□□□□’, ‘□□□□□□’, ‘□□□□’, ‘□□□□□□□□’, ‘□□□□□□□□’, ‘□□□□□’, ‘□□□□□□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□□□’, ‘□□□□’, ‘□□□□□□□’, ‘□□□□’, ‘□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□□’, ‘□□□□□□□□□□’, ‘□□□□’, ‘□□□□□□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□’, ‘□□□□□’, ‘□□□□□’, ‘□□□□□□□□’, ‘□□□□□□□’, ‘□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□□□□□’, ‘□□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□’, ‘□□□□□□□’, ‘□□□□’, ‘□□□□□□□□□□□□’, ‘□□□□□□□□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□□□□□□□□’, ‘□□□□□’, ‘□□□□□□□□’, ‘□□□□’, ‘□□□□’, ‘□□□□□□□□□□□□’,

‘□□□’, ‘□□□□’, ‘□□□□□□□’, ‘□□□’, ‘□□□□’, ‘□□□’, ‘□□□’, ‘□□□□□’, ‘□□□□□□□□□’, ‘□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□□□□□□□’, ‘□□□□□’, ‘□□□’, ‘□□□□□□□’, ‘□□□□□’, ‘□□□□□□’, ‘□□□□□□□□□□□□□’, ‘□□□’, ‘□□□’, ‘□□□□□’, ‘□□□□□’, ‘□□□□’, ‘□□□□’, ‘□□□’]

##

  ,   `pakages`

```shell
pk.py
fibp
   cal
      cal.py
   pt
       pt.py
```

(नोट: कोड ब्लॉक्स और फ़ाइल/डायरेक्टरी नामों को अनुवादित नहीं किया जाता है।)

cal.py:

```
def f(n):
    if n < 2:
        return n
    else:
        return f(n-1) + f(n-2)

def fl(n):
    return list(map(f, range(5)))
```

(यह कोड पायथन में लिखा गया है और इसे हिंदी में अनुवाद करने की आवश्यकता नहीं है।)

pt.py:

```
def p(l):
    print(l, end=' ')

def pln(l):
    print(l)
```

□□.□□:

```
import fibp.cal.cal
import fibp.pt.pt

□□□□.□□.□□.□(□□□□.□□□.□□□.□□(10))
```

`pk.py`   :

```python
from fibp.cal import cal
from fibp.pt import pt

□□.□(□□□.□□(10)) “‘
```