3. Python Facts

이번 시간에는 Python이 갖는 언어적/기능적 특성들을 살펴보고, 이러한 특성들이 개발에 어떠한 편의를 제공할 수 있는지 코드를 통해 살펴보겠습니다.

(1) In Python, everything is done by reference. It doesn't support pointers.

가장 먼저 Python에는 명시적으로 표현된 포인터가 존재하지 않습니다 (yay!). 이는 안전한 실행환경과 상대적으로 덜 복잡한 개발환경을 제공한다는 점에서 장점이 될 수 있습니다.

명시적인 포인터를 제공하지는 않지만, Python 내부적으로 모든 자료형은 object의 형태로 존재하며, 함수 간 arguments를 전달할 때에도 call-by-reference의 방식으로 요청이 이루어집니다. 이러한 자료형 측면의 특징들은 Java와 매우 유사하다고 생각할 수 있습니다.

(2) One function can return multiple values in Python.

Python의 함수들은 하나 이상의 return 값을 가질 수 있습니다. 이는 상황에 따라 함수에 붙는 arguments를 줄여 코드를 단순화하고, 전역변수 사용을 억제하는 효과를 갖습니다.

```
# Multiple Return Values in Python!
def func():
   return 1, 2, 3, 4, 5

one, two, three, four, five = func()
print(one, two, three, four, five)

a, _, _, b, c = func()
print(a, b, c)

```

```
1 2 3 4 5
1 4 5
```

(3) One can use an "else" clause with a "for" loop in Python. It's a special type of syntax that executes only if the for loop exits naturally, without any break statements.

Python에서는 for block 뒤에 else block을 추가할 수 있습니다. 추가된 else block은 for 구문의 종료 조건(stopping condition)에 의해 반복이 중단될 때 수행됩니다.

```
def func(array):
  for num in array:
    if num%2==0:
      print(num)
      break # Case1: Break is called, so 'else' wouldn't be executed.
  else: # Case 2: 'else' executed since break is not called
    print("No call for Break. Else is executed")

print("1st Case:")
a = [2]
func(a)
print("2nd Case:")
a = [1]
func(a)

```

```
1st Case:
2
2nd Case:
No call for Break. Else is executed
```

(4) Function Argument Unpacking is another awesome feature of Python. One can unpack a list or a dictionary as function arguments using * and ** respectively. This is commonly known as the Splat operator.

Array의 개별 요소(element)마다 index를 지정하거나 회람(traverse)을 하면서 귀찮음을 느꼈던 적이 있나요? Python은 그러한 귀찮은 일들을 "알아서" 처리해줍니다.

```
def point(x, y):
  print(x,y)

foo_list = (3, 4)
bar_dict = {'y': 3, 'x': 2}

point(*foo_list) # Unpacking Lists
point(**bar_dict) # Unpacking Dictionaries

colors = ['red', 'green', 'blue', 'yellow', 'cyan', 'purple']
print(*colors)
```

```
3 4
2 3
red green blue yellow cyan purple
```

(5) Want to find the index inside a for loop? Wrap an iterable with 'enumerate' and it will yield the item along with its index.

Enumerate()을 사용하여 iterator를 만들고, 이것을 이용하여 list의 요소들을 traverse하는 동시에 index를 부여할 수 있습니다. 이 기능은 데이터 기준의 iteration 코드 작성시 유용하게 사용될 수 있습니다.

```
# Know the index faster
vowels=['a','e','i','o','u']
for i, letter in enumerate(vowels):
  print (i, letter)

```

```
0 a
1 e
2 i
3 o
4 u
```

(6) One can chain comparison operators in Python; answer= 1<x<10 is executable in Python.

Python에서는 "if 1<x and x<10:"과 "if 1<x<10:" 두 조건절의 표현이 모두 사용 가능합니다.

```
# Chaining Comparison Operators
i = 5;

ans = 1 < i < 10
print(ans)

ans = 10 > i <= 9
print(ans)

ans = 5 == i
print(ans)

```

```
True
True
True
```

(7) Python can define Infinities.

Python에서는 특별한 라이브러리를 사용하지 않아도 infinity을 기술하고 사용할 수 있습니다. 이는 다양한 문맥에서 코드를 단순화하는데 도움이 됩니다.

```
# Positive Infinity
p_infinity = float('Inf')

if 99999999999999 > p_infinity:
  print("The number is greater than Infinity!")
else:
  print("Infinity is greatest")

# Negetive Infinity
n_infinity = float('-Inf')
if -99999999999999 < n_infinity:
  print("The number is lesser than Negative Infinity!")
else:
  print("Negative Infinity is least")

```

```
Infinity is greatest
Negative Infinity is least
```

(8) Instead of building a list with a loop, one can build it more concisely with a *`list comprehension`*.

List comprehension이라는 기능을 활용하여 어떠한 프로그래밍 문법보다도 편리하고, 간결하게 list를 작성할 수 있습니다.

```
# Simple List Append
a = []
for x in range(0,10):
  a.append(x)
print(a)

# List Comprehension
print([x for x in a])

```

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

(9) Python's special Slice Operator is a way to get items from lists, as well as change them.

List 형태의 데이터 요소 액세스를 위해 Python은 확장된 index 개념을 소개합니다. 바로 Slicing이라는 개념입니다. 다음 예제를 통해 다양한 slicing 방법들을 살펴보세요.

```
# Slice Operator
a = [1,2,3,4,5]

print(a[0:2]) # Choose elements [0-2), upper-bound noninclusive

print(a[0:-1]) # Choose all but the last

print(a[::-1]) # Reverse the list

print(a[::2]) # Skip by 2

print(a[::-2]) # Skip by -2 from the back

```

```
[1, 2]
[1, 2, 3, 4]
[5, 4, 3, 2, 1]
[1, 3, 5]
[5, 3, 1]
```

(10) There is actually a poem written by Tim Peters named as THE ZEN OF PYTHON which can be read by just writing import this in the interpreter.

Python 라이브러리 패키지 중에는 한 편의 시를 담고 있는 모듈도 포함되어 있습니다. 지금 코드 창에서 `import this`를 해보세요.

```
# Try to guess the result before you actually run it
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

References

Harshit Gupta. "*Py-Facts – 10 interesting facts about Python*." https://www.geeksforgeeks.org/py-facts-10-interesting-facts-about-python/