

Python Basics

Prof. Gheith Abandah

Reference

- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and Ipython, O'Reilly Media, 2nd Edition, 2018.
 - Material: <https://github.com/wesm/pydata-book>
- Vanderplas, Jacob T. **A Whirlwind Tour of Python**. O'Reilly Media, 2016.
<https://www.oreilly.com/programming/free/files/a-whirlwind-tour-of-python.pdf>
 - <https://github.com/jakevdp/WhirlwindTourOfPython/>

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Quick Python Syntax

- **Comments** are marked by **#**.
- **Quotation marks** (" ') can also be used to enter comments.
- Use **** to **extend a statement** on the next line.
- Semicolon **;** can optionally **terminate a statement**.

```
x += 2 # shorthand for x = x + 2
```

```
# Comments
```

```
"""
```

```
Multi-line comment often  
used in documentation
```

```
"""
```

```
"Single-line Comment"
```

```
In [2]: x = 1 + 2 + 3 + 4 +\  
        5 + 6 + 7 + 8
```

```
lower = []; upper = []
```

Quick Python Syntax

- In Python, **code blocks** are denoted by **indentation**.
- **Four spaces** are usually used.
- Which code snippet always prints **x**?

```
for i in range(10):  
    if i < midpoint:  
        lower.append(i)  
    else:  
        upper.append(i)
```

```
>>> if x < 4:  
...     y = x * 2  
...     print(x)  
  
>>> if x < 4:  
...     y = x * 2  
...     print(x)
```

Quick Python Syntax

- **Parentheses** are for:
 - **Grouping**
 - **Calling**

```
In [5]: 2 * (3 + 4)
Out [5]: 14
```

```
In [6]: print('first value:', 1)
first value: 1
In [7]: print('second value:', 2)
second value: 2
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Variables and Objects

- Python **variables** are **pointers** to **objects**.
- **Variable names** can point to objects of **any type**.

```
x = 1          # x is an integer  
x = 'hello'   # now x is a string  
x = [1, 2, 3] # now x is a list
```


Variables and Objects

- If we have two variable names pointing to the same mutable object, then changing one will change the other as well!

```
In [2]: x = [1, 2, 3]
        y = x
```

```
In [3]: print(y)
```

```
[1, 2, 3]
```

```
In [4]: x.append(4) # append 4 to the list pointed to by x
        print(y) # y's list is modified as well!
```

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

```
In [5]: x = 'something else'
        print(y) # y is unchanged
```

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

Variables and Objects

- Numbers, strings, and other **simple types** are **immutable**.

```
In [6]: x = 10
        y = x
        x += 5 # add 5 to x's value, and assign it to x
        print("x =", x)
        print("y =", y)
```

```
x = 15
```

```
y = 10
```

Variables and Objects

- **Everything is an object**
- Object have **attributes** and **methods** accessible through the dot syntax (.)

```
In [10]: L = [1, 2, 3]
         L.append(100)
         print(L)

[1, 2, 3, 100]
```

```
In [7]: x = 4
        type(x)

Out [7]: int

In [8]: x = 'hello'
        type(x)

Out [8]: str

In [9]: x = 3.14159
        type(x)

Out [9]: float
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Arithmetic Operators

Operator	Name	Description
$a + b$	Addition	Sum of a and b
$a - b$	Subtraction	Difference of a and b
$a * b$	Multiplication	Product of a and b
a / b	True division	Quotient of a and b
$a // b$	Floor division	Quotient of a and b, removing fractional parts
$a \% b$	Modulus	Remainder after division of a by b
$a ** b$	Exponentiation	a raised to the power of b
-a	Negation	The negative of a
+a	Unary plus	a unchanged (rarely used)

```
>>> a = 5
>>> b = 3
>>> a / b
1.6666666666666667
>>> a // b
1
>>> a % b
2
```

Bitwise Operators

Operator	Name	Description
<code>a & b</code>	Bitwise AND	Bits defined in both a and b
<code>a b</code>	Bitwise OR	Bits defined in a or b or both
<code>a ^ b</code>	Bitwise XOR	Bits defined in a or b but not both
<code>a << b</code>	Bit shift left	Shift bits of a left by b units
<code>a >> b</code>	Bit shift right	Shift bits of a right by b units
<code>~a</code>	Bitwise NOT	Bitwise negation of a

	8	4	2	1	← Weight
0	0	0	0	0	
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	
10	1	0	1	0	
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	
14	1	1	1	0	
15	1	1	1	1	

```
>>> a = 1
>>> b = 2
>>> print( a & b , a | b , a ^ b , b << a , b >> a , ~b )
0 3 3 4 1 -3
```

Comparison Operators

- Return Boolean values **True** or **False**

Operation	Description
<code>a == b</code>	a equal to b
<code>a != b</code>	a not equal to b
<code>a < b</code>	a less than b
<code>a > b</code>	a greater than b
<code>a <= b</code>	a less than or equal to b
<code>a >= b</code>	a greater than or equal to b

```
>>> a = 1
>>> b = 2
>>> print( a == b , a != b , a < b , a > b )
False True True False
```

Assignment Operators

- Assignment is evaluated from **left to right**.

```
>>> i = j = k = 10
>>> print( i , j , k )
10 10 10
```

- There is an **augmented assignment** operator corresponding to each of the binary arithmetic and bitwise operators.

```
a += b   a -= b   a *= b   a /= b
a //= b  a %= b   a **= b  a &= b
a |= b   a ^= b   a <<= b  a >>= b
```

```
>>> a = 2
>>> b = 10
>>> b += a
>>> print( a , b )
2 12
```


Boolean Operators

- The **Boolean operators** operate on Boolean values:
 - **and**
 - **or**
 - **not**
- Can be used to construct **complex comparisons**.

```
In [15]: x = 4
          (x < 6) and (x > 2)

Out [15]: True

In [16]: (x > 10) or (x % 2 == 0)

Out [16]: True

In [17]: not (x < 6)

Out [17]: False
```

Identity and Membership Operators

Operator	Description
<code>a is b</code>	True if a and b are identical objects
<code>a is not b</code>	True if a and b are not identical objects
<code>a in b</code>	True if a is a member of b
<code>a not in b</code>	True if a is not a member of b

```
In [24]: 1 in [1, 2, 3]
```

```
Out [24]: True
```

```
In [25]: 2 not in [1, 2, 3]
```

```
Out [25]: False
```

```
In [19]: a = [1, 2, 3]
         b = [1, 2, 3]
```

```
In [20]: a == b
```

```
Out [20]: True
```

```
In [21]: a is b
```

```
Out [21]: False
```

```
In [22]: a is not b
```

```
Out [22]: True
```

```
In [23]: a = [1, 2, 3]
         b = a
         a is b
```

```
Out [23]: True
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Python Scalar Types

Type	Example	Description
int	x = 1	Integers (i.e., whole numbers)
float	x = 1.0	Floating-point numbers (i.e., real numbers)
complex	x = 1 + 2j	Complex numbers (i.e., numbers with a real and imaginary part)
bool	x = True	Boolean: True/False values
str	x = 'abc'	String: characters or text
NoneType	x = None	Special object indicating nulls

```
>>> print( int('1') , float(1) , len(str(10)) )  
1 1.0 2
```

Integers and Floats

- **Integers** are variable-precision, **no overflow** is possible.

```
>>> 2 ** 90
1237940039285380274899124224
```

- The **floating-point** type can store fractional numbers. They can be defined either in standard **decimal notation** or in **exponential notation**.

```
In [5]: x = 0.000005
        y = 5e-6
        print(x == y)
```

True

```
In [6]: x = 1400000.00
        y = 1.4e6
        print(x == y)
```

True

Strings

- Strings in Python are created with **single** or **double quotes**.
- The built-in function **len()** returns the string length.
- Any character in the string can be accessed through its **index**.

```
>>> s1 = "Hi "  
>>> s2 = 'Python'  
>>> print( s1 + s2 , len( s2 ) , 3 * s1 , s2[0] )  
Hi Python 6 Hi Hi Hi P
```

None and Boolean

- **Functions that do not return** value return **None**.
- **None** variables are evaluated to **False**.
- The **Boolean type** is a simple type with two possible values: **True** and **False**.
- Values are evaluated to **True** unless they are **None**, zero or empty.

```
>>> print( bool(1.5) , bool(0) , bool(None) , bool([]) )  
True False False False
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Built-In Data Structures

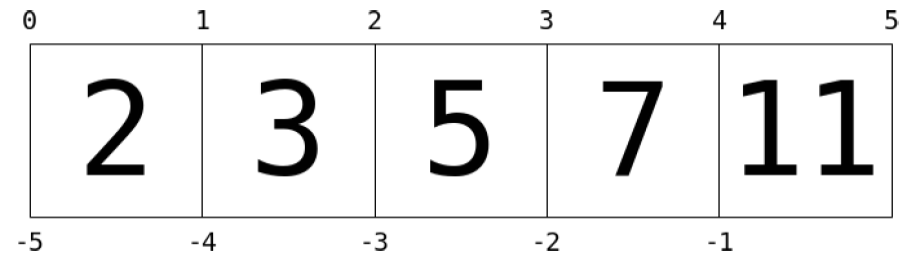
- There are four built in Python data structures.

Type Name	Example	Description
list	[1, 2, 3]	Ordered collection
tuple	(1, 2, 3)	Immutable ordered collection
dict	{'a':1, 'b':2, 'c':3}	Unordered (key,value) mapping
set	{1, 2, 3}	Unordered collection of unique values

Lists

- Lists are **ordered** and **mutable**.
- A list can hold objects of **any type**.
- Python uses **zero-based** indexing.
- Elements at the **end** of the list can be accessed with negative numbers, starting from **-1**.

```
>>> L = [2, 3, 5, 7]
>>> L.append(11)
>>> print( len(L) , L[0] , L[4] )
5 2 11
>>> L = [15, 16] + L
>>> print( L )
[15, 16, 2, 3, 5, 7, 11]
>>> L.sort()
>>> print( L )
[2, 3, 5, 7, 11, 15, 16]
>>> L = [1, 'two', 3.14, [0, 3, 5]]
>>> L[3]
[0, 3, 5]
>>> L[-2]
3.14
```



Lists

- **Slicing** is a means of accessing multiple values in sub-lists.
[start : end+1 : inc]
- **Negative step** reverses the list.
- Both indexing and slicing can be used **to set elements** as well as access them.

```
>>> L = [2, 3, 5, 7, 11]
>>> L[:]
[2, 3, 5, 7, 11]
>>> L[:3]
[2, 3, 5]
>>> L[2:]
[5, 7, 11]
>>> L[1:4]
[3, 5, 7]
>>> L[::2]
[2, 5, 11]
>>> L[::-1]
[11, 7, 5, 3, 2]
>>> L[0] = 100
>>> L[1:3] = [20, 30]
>>> L
[100, 20, 30, 7, 11]
```

Tuples

- Tuples are **similar to lists**, but are **immutable**.
- Can be defined with or without parentheses **()**.
- **Functions return** multiple values as **tuples**.

```
>>> t = (1, 2, 3)
>>> t = 1, 2, 3
>>> print( t[2], len(t) )
3 3
>>> x = 0.25
>>> x.as_integer_ratio()
(1, 4)
>>> numerator, denominator = x.as_integer_ratio()
>>> print( numerator, denominator )
1 4
```

Dictionaries

- Dictionaries are flexible **mappings** of **keys** to **values**.
- They can be created via a **comma-separated list** of **key:value** pairs within **curly braces**.

```
>>> d = {'Name':'Sami', "Weight":75}
>>> d['Length'] = 1.75
>>> d
{'Name': 'Sami', 'Weight': 75, 'Length': 1.75}
>>> d['Name']
'Sami'
```

Sets

- Sets are **unordered** collections of **unique** items.
- They are defined using curly brackets **{ }**.
- Set operations include **union**, **intersection**, **difference** and **symmetric difference**.

```
>>> primes = {2, 3, 5, 7}
>>> odds = {1, 3, 5, 7, 9}
>>> primes | odds      # Union
{1, 2, 3, 5, 7, 9}
>>> primes & odds      # Intersection
{3, 5, 7}
>>> primes - odds      # Differences
{2}
>>> primes ^ odds      # Symmetric difference
{1, 2, 9}
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Conditional Statements: `if`, `elif`, and `else`

- `if` statements in Python have optional `elif` and `else` parts.

```
In [1]: x = -15

if x == 0:
    ← print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")

-15 is negative
```


for Loops

- The **for** loop is repeated for each index returned by the **iterator** after **in**.

```
In [2]: for N in [2, 3, 5, 7]:  
        ← print(N, end=' ') # print all on same line  
  
2 3 5 7
```

- The **range()** object is very useful in for loops.

```
In [3]: for i in range(10):  
        print(i, end=' ')  
  
0 1 2 3 4 5 6 7 8 9
```

for Loops

- The `range(start, end+1, inc)` has default zero start and unit increment.

```
In [4]: # range from 5 to 10  
        list(range(5, 10))
```

```
Out [4]: [5, 6, 7, 8, 9]
```

```
In [5]: # range from 0 to 10 by 2  
        list(range(0, 10, 2))
```

```
Out [5]: [0, 2, 4, 6, 8]
```

while Loops

- The **while** loop iterates as long as the condition is met.

```
In [6]: i = 0
        while i < 10:
            ← print(i, end=' ')
            i += 1
```

0 1 2 3 4 5 6 7 8 9

break and continue: Fine-Tuning Your Loops

- The **continue** statement skips the remainder of the current loop, and goes to the next iteration.

```
In [7]: for n in range(20):  
        # check if n is even  
        if n % 2 == 0:  
            continue  
        print(n, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

Prints odd
numbers

break and continue: Fine-Tuning Your Loops

- The **break** statement breaks out of the loop entirely.

```
In [8]: a, b = 0, 1
        amax = 100
        L = []

        while True:
            (a, b) = (b, a + b)
            if a > amax:
                break
            L.append(a)

        print(L)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

List all Fibonacci numbers up to 100.

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators

Defining Functions

- Functions are defined with the **def** statement.
- The following function returns a list of the first N Fibonacci numbers.

```
In [4]: def fibonacci(N):  
        ↔ L = []  
        a, b = 0, 1  
        while len(L) < N:  
            a, b = b, a + b  
            L.append(a)  
        return L
```

- Calling it:

```
In [5]: fibonacci(10)  
Out [5]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Default Argument Values

- You can have **default values** for arguments.

```
def fibonacci(N, a=0, b=1):  
    L = []  
    while len(L) < N:  
        a, b = b, a + b  
        L.append(a)  
    return L
```

- It can be called with or without the optional args.

```
fibonacci(10)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
fibonacci(10, 0, 2)
```

```
[2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```


*args and **kwargs: Flexible Arguments

- Functions can be defined using ***args** and ****kwargs** to capture variable numbers of **arguments** and **keyword arguments**.

```
In [11]: def catch_all(*args, **kwargs):  
         print("args =", args)  
         print("kwargs = ", kwargs)
```

```
In [12]: catch_all(1, 2, 3, a=4, b=5)
```

```
args = (1, 2, 3)  
kwargs = {'a': 4, 'b': 5}
```

Tuple

Dictionary

```
In [13]: catch_all('a', keyword=2)
```

```
args = ('a',)  
kwargs = {'keyword': 2}
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- **Objects and Classes**
- **Errors and Exceptions**
- **Iterators**
- **List Comprehensions**
- **Generators**

Objects and Classes

- Python is **object-oriented** programming language.
- **Objects** bundle together **data** and **functions**.
- Each Python object has a type, or **class**.
- An object is an **instance** of a class.
- Accessing **instance data**:
`object.attribute_name`
- Accessing **instance methods**:
`object.method_name(parameters)`

String Objects

- String objects are instances of class **str**.

```
name = input("Please enter your name: ")  
print("Hello " + name.upper() + ", how are you?")
```

```
Please enter your name: Sami  
Hello SAMI, how are you?
```

- String objects have many **useful methods**

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

String Methods

```
>>> s = "  Hi  "  
>>> s.strip()  
'Hi'  
>>> 'Age: {0}, Weight: {1}'.format(20, 70)  
'Age: 20, Weight: 70'  
>>> s = 'This is a string'  
>>> s.find('is')  
2  
>>> s.replace('a', 'the')  
'This is the string'
```

String Objects

- Accept the **escape character** `\`.
- **Unicode** encoded.

```
s = 'The cat\'s tail \n is \t long.'  
print(s)
```

```
The cat's tail  
is      long.
```

```
s = 'بايثون'  
print(s)
```

```
بايثون
```

```
s_utf8 = s.encode('utf-8')  
print(s_utf8)
```

```
b' \xd8\xa8\xd8\xa7\xd9\x8a\xd8\xab\xd9  
\x88\xd9\x86 '
```

Date and Time Objects

- The built-in Python **datetime** module provides **datetime**, **date**, and **time** types.
- Such objects can be **formatted** and accept **-** and **+** operands.

```
from datetime import datetime, date, time
dt = datetime(1999, 8, 16, 8, 30, 0)
print(dt.day)
```

```
16
```

```
dt2 = datetime(2000, 8, 16, 8, 30, 0)
delta = dt2 - dt
dt3 = dt2 + delta
print(dt3.strftime('%d/%m/%Y %H:%M'))
```

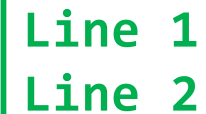
```
17/08/2001 08:30
```

File Objects

- Files can be opened for **read**, **write** or **append**.

```
f = open('myfile.txt', 'w')
f.write('Line 1\n')
f.write('Line 2\n')
f.close()
```

```
f = open('myfile.txt', 'r')
for line in f:
    print(line.strip())
f.close()
```



```
Line 1
Line 2
```


Classes

- New class types can be defined using **class** keyword.

```
class Animal(object):  
    def __init__(self, name='Animal'): # Constructor  
        print('Constructing an animal!')  
        self.name = name  
        if name == 'Cat':  
            self.meows = True # Attribute  
        else:  
            self.meows = False  
        super(Animal, self).__init__()  
  
    def does_meow(self): # Method  
        return self.meows
```

```
cat = Animal('Cat')  
print('It meows ', cat.does_meow())
```

```
Constructing an animal!  
It meows True
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- **Errors and Exceptions**
- **Iterators**
- **List Comprehensions**
- **Generators**

Runtime Errors

1. Referencing an **undefined variable**
2. **Unsupported operation**
3. **Division by zero**
4. Accessing a sequence element that **doesn't exist**

```
In [1]: print(Q)
```

```
In [2]: 1 + 'abc'
```

```
In [3]: 2 / 0
```

```
In [4]: L = [1, 2, 3]  
        L[1000]
```

Catching Exceptions: try and except

- Runtime exceptions can be handled using the **try...except** clause.

```
In [6]: try:
        print("let's try something:")
        x = 1 / 0 # ZeroDivisionError
    except:
        print("something bad happened!")

let's try something:
something bad happened!
```

try..except..else..finally

- Python also support **else** and **finally**

```
In [23]: try:
          print("try something here")
        except:
          print("this happens only if it fails")
        else:
          print("this happens only if it succeeds")
        finally:
          print("this happens no matter what")
```

```
try something here
this happens only if it succeeds
this happens no matter what
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- **Iterators**
- **List Comprehensions**
- **Generators**

Iterators

- Iterators are used in **for** loops and can be used using **next()**

```
In [1]: for i in range(10):  
        print(i, end=' ')
```

0 1 2 3 4 5 6 7 8 9

```
In [4]: I = iter([2, 4, 6, 8, 10])
```

```
In [5]: print(next(I))
```

2

```
In [6]: print(next(I))
```

4

Iterators

- The range iterator
- Iterating over lists
- **enumerate** iterator

```
In [1]: for i in range(10):  
        print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

```
In [2]: for value in [2, 4, 6, 8, 10]:  
        # do some operation  
        print(value + 1, end=' ')
```

```
3 5 7 9 11
```

```
In [14]: for i, val in enumerate(L):  
         print(i, val)
```

```
0 2
```

```
1 4
```

```
2 6
```

```
3 8
```

```
4 10
```


Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- **List Comprehensions**
- **Generators**

List Comprehensions

- A way to compress a **list-building** for loop into a single short, readable line.
- Syntax: ***[expr for var in iterable]***

```
In [3]: [n ** 2 for n in range(12)]
```

```
Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

```
In [1]: [i for i in range(20) if i % 3 > 0]
```

```
Out [1]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

```
In [4]: [(i, j) for i in range(2) for j in range(3)]
```

```
Out [4]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

List Comprehensions

- Lists comprehensions can be used to construct **sets** with no duplicates.

```
In [10]: {a % 3 for a in range(1000)}  
Out [10]: {0, 1, 2}
```

- Or **dictionaries**

```
In [11]: {n:n**2 for n in range(6)}  
Out [11]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Outline

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- **Generators**

Generators

- A list is a collection of values, while a **generator expression** is a **recipe** for producing values.

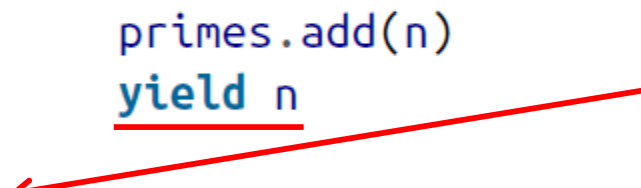
```
In [5]: G = (n ** 2 for n in range(12))
        for val in G:
            print(val, end=' ')
0 1 4 9 16 25 36 49 64 81 100 121
```

Generators

- A **generator function** uses **yield** to yield a sequence of values.

```
In [19]: def gen_primes(N):  
         """Generate primes up to N"""  
         primes = set()  
         for n in range(2, N):  
             if all(n % p > 0 for p in primes):  
                 primes.add(n)  
                 yield n  
  
         print(*gen_primes(70))  
  
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
```

Get a sequence from
the generator



Homework

- Solve the homework on **Python Basic Programming**

Summary

- Quick Python Syntax
- Variables and Objects
- Operators
- Built-In Types: Simple Values
- Built-In Data Structures
- Control Flow
- Defining and Using Functions
- Objects and Classes
- Errors and Exceptions
- Iterators
- List Comprehensions
- Generators