

Python & Git Fundamentals

University of Stavanger

DATPREP

Based on “A Whirlwind Tour of Python” by Jake VanderPlas (O’Reilly)

Goal

- Understand what Python is
- Review basic Python syntax
 - Variables
 - Operations
 - Useful Built-ins
 - Control Flows
 - Functions
- Tasks

What is Python?

- High-level programming language
- Interpreted – not compiled
- Object oriented
- Dynamic typing
- Why choose Python?
 - Easy to read and write
 - Main target for many popular data science libraries
 - Widely used

Disadvantages with Python

- More difficult to create large applications
- No compiler – will not find obvious type mismatches before execution
- Interpreted language – often slower than compiled languages

Basic Syntax Example

```
# set the midpoint
midpoint = 5

# make two empty lists
lower = []; upper = []

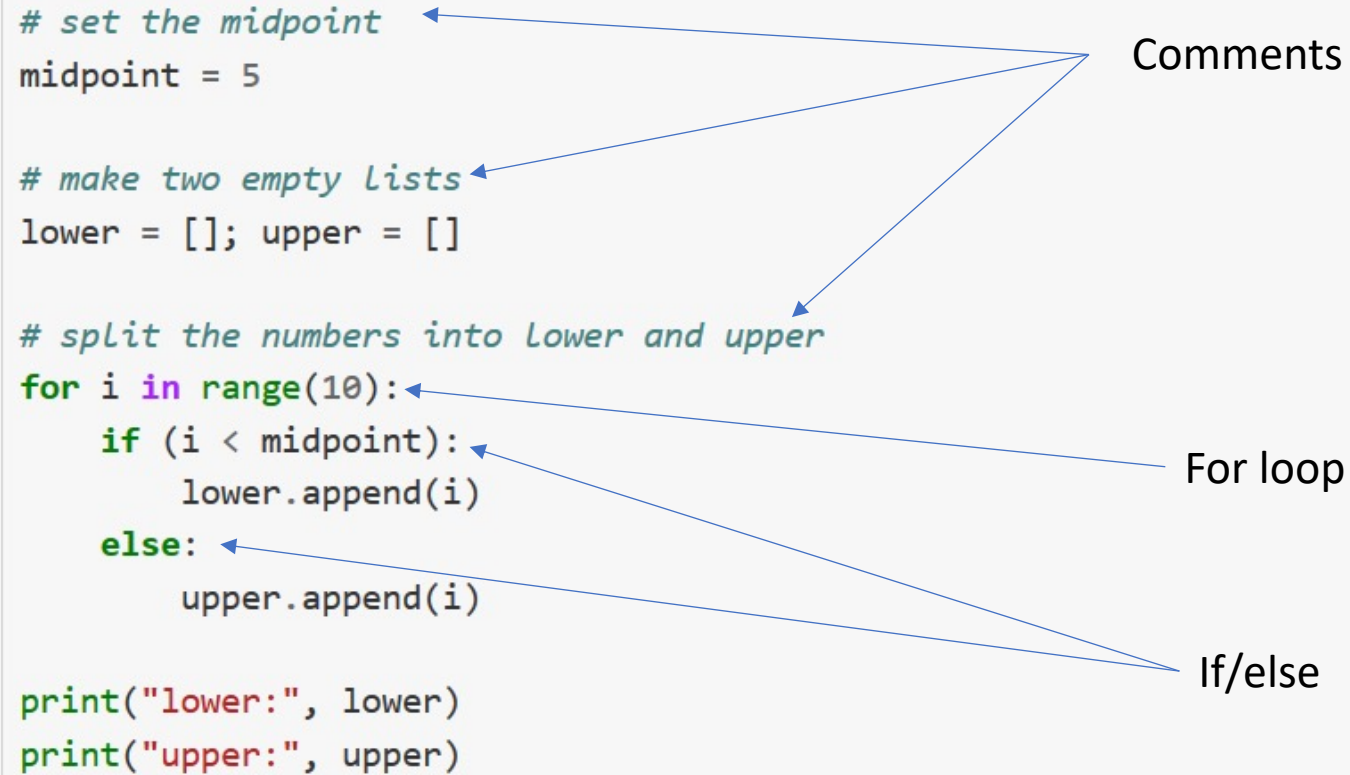
# split the numbers into lower and upper
for i in range(10):
    if (i < midpoint):
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)
```

Comments

For loop

If/else



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

Note: No; at the end of each line, indentation determines block, not { and }

Variables

- Assigning Variables

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

- Everything is an object

```
x = 4
type(x)
```

int

```
x = 'hello'
type(x)
```

str

```
x = 3.14159
type(x)
```

float

Variables

- Most types have inbuilt functions

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

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

- Be careful when assigning variables to other variables

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

```
print(y)
```

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

```
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]
```

```
x = 'something else'
print(y) # y is unchanged
```

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

Operations - Arithmetic

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

```
# addition, subtraction, multiplication  
(4 + 8) * (6.5 - 3)
```

42.0

Operations – Bitwise

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

Operations – Assignment Shorthand

- One for each operation covered

```
a += b
```

```
a -= b a *= b
```

```
a /= b
```

```
a // = b a %= b a ** = b a &= b
```

```
a |= b
```

```
a ^= b a <<= b a >>= b
```

Operations – Comparison

OperationDescription

`a == b` `a` equal to `b`

`a < b` `a` less than `b`

`a <= b` `a` less than or equal to `b`

OperationDescription

`a != b` `a` not equal to `b`

`a > b` `a` greater than `b`

`a >= b` `a` greater than or equal to `b`

```
# 25 is odd  
25 % 2 == 1
```

True

```
# 66 is odd  
66 % 2 == 1
```

False

Operations – Boolean

- Combining and negating comparisons

```
x = 4  
(x < 6) and (x > 2)
```

True

```
(x > 10) or (x % 2 == 0)
```

True

```
not (x < 6)
```

False

Operations – Identity and Membership

Operator	Description
----------	-------------

<code>a is b</code>	True if <code>a</code> and <code>b</code> are identical objects
---------------------	---

<code>a is not b</code>	True if <code>a</code> and <code>b</code> are not identical objects
-------------------------	---

<code>a in b</code>	True if <code>a</code> is a member of <code>b</code>
---------------------	--

<code>a not in b</code>	True if <code>a</code> is not a member of <code>b</code>
-------------------------	--

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

True

```
2 not in [1, 2, 3]
```

False

Useful Built-ins – Simple 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 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

Useful Built-ins – 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

Useful Built-ins – Lists

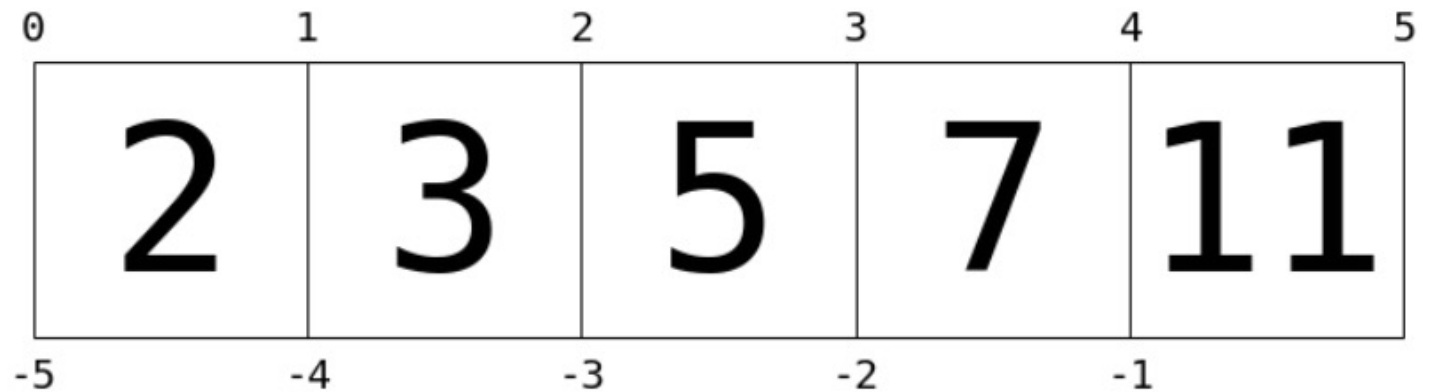
```
L = [2, 3, 5, 7, 11]
```

```
L[0]
```

2

```
L[-1]
```

11



Useful Built-ins – List Slicing

```
L = [2, 3, 5, 7, 11]
```

```
L[:3]
```

```
[2, 3, 5]
```

```
L[-3:]
```

```
[5, 7, 11]
```

```
L[0] = 100
```

```
print(L)
```

```
[100, 3, 5, 7, 11]
```

```
L[1:3] = [55, 56]
```

```
print(L)
```

```
[100, 55, 56, 7, 11]
```

Useful Built-ins – List Slicing Trick

```
L = [2, 3, 5, 7, 11]
```

```
L[::2] # equivalent to L[0:len(L):2]
```

```
[2, 5, 11]
```

Third value = step

```
L[::-1]
```

```
[11, 7, 5, 3, 2]
```

Reversed list!

Useful Built-ins – Dictionaries

```
numbers = {'one':1, 'two':2, 'three':3}
```

```
# Access a value via the key  
numbers['two']
```

2

```
# Set a new key:value pair  
numbers['ninety'] = 90  
print(numbers)
```

```
{'three': 3, 'ninety': 90, 'two': 2, 'one': 1}
```

Control Flows – if, elif, else

```
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

Control Flows – for Loops

```
for i in range(10):  
    print(i, end=' ')
```

0 1 2 3 4 5 6 7 8 9

```
# range from 5 to 10  
list(range(5, 10))
```

[5, 6, 7, 8, 9]

```
# range from 0 to 10 by 2  
list(range(0, 10, 2))
```

[0, 2, 4, 6, 8]

Control Flows – while Loops

- Less often used than for loops

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1
```

0 1 2 3 4 5 6 7 8 9

Control Flows – break and continue

```
for n in range(20):  
    # if the remainder of n / 2 is 0, skip the rest of the loop  
    if n % 2 == 0:  
        continue  
    print(n, end=' ')
```

Continues to next iteration if n is even

1 3 5 7 9 11 13 15 17 19

```
a, b = 0, 1  
amax = 100  
L = []  
  
while True:  
    (a, b) = (b, a + b)  
    if a > amax:  
        break  
    L.append(a)  
  
print(L)
```

Exits the loop if a > 100

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

Calling Functions

Arguments

```
print(1, 2, 3)
```

1 2 3

Keyword Argument – must be at the end

```
print(1, 2, 3, sep='--')
```

1--2--3

Defining Functions

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

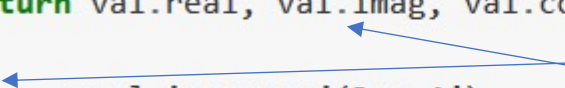
```
fibonacci(10)
```

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

```
def real_imag_conj(val):  
    return val.real, val.imag, val.conjugate()
```

```
r, i, c = real_imag_conj(3 + 4j)  
print(r, i, c)
```

Multiple return values



```
3.0 4.0 (3-4j)
```

Defining Functions – Default Values

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

```
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]
```

```
fibonacci(10, b=3, a=1)
```

```
[3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

Defining Functions – Flexible Arguments

```
def catch_all(*args, **kwargs):  
    print("args =", args)  
    print("kwargs = ", kwargs)
```

```
catch_all(1, 2, 3, a=4, b=5)
```

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

```
catch_all('a', keyword=2)
```

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

```
inputs = (1, 2, 3)  
keywords = {'pi': 3.14}  
  
catch_all(*inputs, **keywords)
```

```
args = (1, 2, 3)  
kwargs = {'pi': 3.14}
```

Tasks!

You will continue developing the concepts we have gone through