Lecture 10

Introduction to Python. Built-in data types. Variables, lists, tuples, dictionaries, sets. Control structures. Functions. Docstrings.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

30 Nov 2023

Outline

1. Introduction
2. Built-in data types
    - Numeric, Boolean, strings
    - Lists and tuples
    - Sets and dictionaries
3. Control structures
4. Comprehensions
5. Exceptions
6. Functions
7. Docstrings

Introduction

Why Python?

- **Readability and simplicity:** Python's syntax is designed for readability, making it easy for beginners and promoting clean code in large projects.
- **Versatility:** Supports various programming paradigms (procedural, OO, functional).
- **Extensive standard library:** Comes with a comprehensive standard library, reducing the need for writing code from scratch and promoting code reusability.
- **Large and active community:** Boasts a vibrant community, fostering collaboration, and providing a wealth of third-party libraries and online resources.
- **Data Science and Machine Learning:** Preferred language for data science and machine learning with powerful libraries like NumPy, SciPy, Pandas, PyTorch, and TensorFlow.
- **Cross-platform compatibility:** Code written in Python can run on different operating systems without modification.
- **Industry adoption:** Widely adopted in countless startups and enterprises.

Setting up a Python environment

To work with Python, you need to set up a development environment.

Here are the basic steps:

- **Install Python:** Download and install Python (version ≥ 3) from the official Python website. Advanced users may want to have a look at PyPy.
- **Integrated Development Environment (IDE):** Choose an IDE such as PyCharm, VSCode, or Jupyter Notebook for a more interactive development experience. You can even use online platforms like Google Colab and JupyterLab.
- **Package management:** Utilize tools like `pip` to install and manage third-party packages.
- (Advanced users) **Virtual environments:** Use virtual environments, such as conda to isolate project dependencies and avoid conflicts between different projects.

Structure of a basic Python program

A Python program typically consists of the following components:

- **Comments:** Lines starting with `#` are comments. They are ignored by the Python interpreter and serve as notes for developers.
- **Statements:** Python code is composed of statements, which are instructions that the interpreter can execute.
- **Indentation:** Python uses indentation to define code blocks. Consistent indentation is crucial for proper program structure.

Hello, world!

```
# This is a comment.
print("Hello, World!")  # This is also a comment.

x = 3

# Code block with proper indentation.
if x > 0:
    print("Positive")
```

Definitions

A **value** is a piece of data that a computer program works with such as a number or text. There are different **types** of values: `42` is an integer and `"Hello!"` is a string. A **variable** is a name that refers to a value. In mathematics and statistics, we usually use variable names like x and y. In Python, we can use any word as a variable name as long as it starts with a letter or an underscore. However, it should not be a reserved word in Python such as `for`, `while`, `class`, `lambda`, etc. as these words encode special functionality in Python that we don't want to overwrite!

It can be helpful to think of a variable as a box that holds some information (a single number, a vector, a string, etc). We use the **assignment operator** `=` to assign a value to a variable.

Built-in data types

Python as a strongly, dynamically typed language

Python typing is **dynamic** so you can change a string variable to an `int` (in a **static** language you can't):

```
x = 'somestring'
x = 50
```

Python typing is **strong** so you can't merge types:

```
'foo' + 3 # TypeError: cannot concatenate 'str' and 'int' objects
```

In weakly-typed languages (such as Javascript) this happens:

```
 'foo' + 3 = 'foo3'
```

Built-in data types (1/2)

See the Python 3 documentation for a summary of the standard built-in Python datatypes.

Type name	Type Category	Description	Example
`int`	Numeric Type	integer number	`42`
`float`	Numeric Type	real number	`3.14159`
`complex`	Numeric Type	complex number	`1.0 + 2.0j`
`bool`	Boolean Values	true or false	`True`
`str`	Sequence Type	text	`"I Can Has Cheezburger?"`

Built-in data types (2/2)

Type name	Type Category	Description	Example
`list`	Sequence Type	a collection of objects - mutable & ordered	`['Ali', 'Xinyi', 'Miriam']`
`tuple`	Sequence Type	a collection of objects - immutable & ordered	`('Thursday', 6, 9, 2018)`
`set`	Set Type	a collection of unique objects - mutable & unordered	`{'jack', 'sjoerd'}`
`dict`	Mapping Type	mapping of key-value pairs	`{'name':'DSAI', 'code':123, 'credits':6}`
`NoneType`	Null Object	represents no value	`None`

Numeric, Boolean, strings

Numeric data types

There are three distinct numeric types: `integers`, `floating point numbers`, and `complex numbers`. We can determine the type of an object in Python using `type()`. We can print the value of the object using `print()`.

```
x = 42
type(x) # int
print(x) # 42

pi = 3.14159
type(pi) # float

z = 1 + 3j # complex
print(abs(z))
import cmath
cmath.phase(z)
```

Arithmetic operators

Below is a table of the syntax for common arithmetic operations in Python:

Operator	Description
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation
`//`	integer division / floor division
`%`	modulo

Arithmetic operators: examples

```
1 + 2 + 3 + 4 + 5 # 15
2 * 3.14159 # 6.28318
2 ** 10 # 1024
```

Division may produce a different `dtype` than expected, it will change `int` to `float`.

```
int_2 = 2
type(int_2) # int
int_2 / int_2 # 1.0
type(int_2 / int_2) # float
```

But the syntax `//` allows us to do *integer division* (aka *floor division*) and retain the `int` data type, it always rounds down.

```
101 / 2 # 50.5
101 // 2 # 50 (floor division: always rounds down).
```

Boolean

The Boolean (`bool`) type has two values: `True` and `False`.

We can compare objects using comparison operators, and we'll get back a Boolean result:

Operator	Description
`x == y`	`x` is equal to `y`
`x != y`	`x` is not equal to `y`
`x > y`	`x` is greater than `y`
`x >= y`	`x` is greater than or equal to `y`
`x < y`	`x` is less than `y`
`x <= y`	`x` is less than or equal to `y`
`x is y`	`x` is the same object as `y`

Boolean operators

We also have so-called *boolean operators* which also evaluate to either `True` or `False`:

Operator	Description: `True` if
`x and y`	both `x` and `y` are `True`
`x or y`	at least one of `x` and `y` is `True`
`not x`	`x` is `False`

Python also has bitwise operators like AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), shift (`<<` and `>>`).

Strings

Strings represent sequences of characters and are widely used in Python.

- **String creation:** Strings can be created using single (`'`) or double (`"`) quotes.
- **String operations:** Concatenation (`+`), repetition (`*`), and indexing.

Example

```
# String creation.
message = "Hello, Python!"

message_twice = message * 2 # "Hello, Python!Hello, Python!"

# String operations.
greeting = "Hello, "
name = "Alice"
full_greeting = greeting + name  # Concatenation.
```

String manipulation

```
# String methods.
message = "Hello, Python!"

# Length of a string.
length = len(message)

# Upper and lower case.
upper_case = message.upper() # Return a new string.
lower_case = message.lower() # Return a new string.

# String formatting.
formatted_message = f"Message: {message}"
```

There are *many* string methods. Check out the documentation.

String manipulation

```
all_caps = "HOW ARE YOU TODAY?"
all_caps.split() # ['HOW', 'ARE', 'YOU', 'TODAY?']
all_caps.count("O") # 3

caps_list = list(all_caps)
"".join(caps_list) # 'HOW ARE YOU TODAY?'
"-".join(caps_list) # 'H-O-W- -A-R-E- -Y-O-U- -T-O-D-A-Y-?'
"".join(caps_list).lower().split(" ") # ['how', 'are', 'you', 'today?']
```

String formatting

Python has ways of creating strings by *filling in the blanks* and formatting them nicely. This is helpful for when you want to print statements that include variables or statements. There are a few ways of doing this but I use and recommend f-strings which were introduced in Python 3.6. All you need to do is put the letter `f` out the front of your string and then you can include variables with curly-bracket notation `{}`.

```
name = "Newborn Baby"
age = 4 / 12
day = 30
month = 7
year = 2023
template_new = f"Hello, my name is {name}. I am {age:.2f} years old. I was born {day}/{month:02}/{year}."
template_new
```

> 'Hello, my name is Newborn Baby. I am 0.33 years old. I was born 30/07/2023.'

See format code options here.

Lists and tuples

Lists and tuples

Lists and tuples are versatile data structures in Python.

- **Lists:** Mutable, ordered collections of items. Elements can be added, removed, or modified.
- **Tuples:** Immutable, ordered collections. Once defined, elements cannot be changed.

Example

```
# List.
numbers = [1, 2, 3, 4, 5]

# Tuple.
coordinates = (2, 3)
```

List operations

```
# List creation.
my_list = [1, 2, "THREE", 4, 0.5]
another_list = [1, "two", [3, 4, "five"], True, None, {"key": "value"}]
numbers = [1, 2, 3, 4, 5]

len(numbers) # 5

# Adding elements.
my_list.append([6, 7]) # [1, 2, "THREE", 4, 0.5, [6, 7]]
my_list.extend([6, 7]) # [1, 2, "THREE", 4, 0.5, 6, 7]

# Removing elements.
numbers.remove(3) # [1, 2, 4, 5]
popped_value = numbers.pop() # popped_value = 5, numbers = [1, 2, 4]
```

You can see the documentation for more list methods .

Tuple operations

```
# Tuple creation.
today = (1, 2, "THREE", 4, 0.5)
coordinates = (2, 3)

# Unpacking.
x, y = coordinates

# Concatenation.
combined = coordinates + (4, 5) # (2, 3, 4, 5)

# Tuple repetition.
repeated = coordinates * 3 # (2, 3, 2, 3, 2, 3)
```

Indexing

We can access values inside a list, tuple, or string using square bracket syntax. Python uses *zero-based indexing*, which means the first element of the list is in position 0, not position 1.

```
my_list = [1, 2, 'THREE', 4, 0.5]
my_list[0] # 1
my_list[2] # 'THREE'
len(my_list) # 5

my_list[5] # IndexError: list index out of range
```

We can use negative indices to count backwards from the end of the list.

```
my_list = [1, 2, 'THREE', 4, 0.5]
my_list[-1] # 0.5
my_list[-2] # 4
```

Slicing

We can use the colon `:` to access a sub-sequence. This is called *slicing*.

```
my_list[1:3] # [2, 'THREE']
```

Note from the above that the **start** of the slice is **inclusive** and the **end** is **exclusive**. So `my_list[1:3]` fetches elements 1 and 2, but not 3.

Strings behave the same as lists and tuples when it comes to indexing and slicing. Remember, we think of them as a *sequence* of characters.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet[0] # 'a'
alphabet[-1] # 'z'
alphabet[-3] # 'x'
alphabet[:5] # 'abcde'
alphabet[12:20] # 'mnopqrst'
```

Sets and dictionaries

Sets (1/2)

Another built-in Python data type is the `set`, which stores an **unordered** list of **unique** items. Being unordered, sets do not record element position or order of insertion and so do not support indexing.

```
s = {2, 3, 5, 11}
{1, 2, 3} == {3, 2, 1} # True
[1, 2, 3] == [3, 2, 1] # False
s.add(2) # Does nothing.
```

Since element are stored unordered, sets can't be indexed:

```
s[0]
```

> TypeError: 'set' object is not subscriptable

Dictionaries

Dictionaries are key-value pairs, allowing efficient data retrieval.

- **Creating dictionaries:** Define key-value pairs using curly braces `{}`.
- **Accessing values:** Retrieve values using keys.

Example

```
# Dictionary creation.
student = {"name": "Alice", "age": 20, "grade": "A"}

student_name = student["name"] # Access value by key.

student["city"] = "New York" # Add new key-value pair.
student["age"] = 21 # Modify value by key.

del student["grade"] # Remove key.
```

Casting

Sometimes we need to explicitly **cast** a value from one type to another. Python tries to do the conversion, or throws an error if it can't.

```
x = 5.0 # float
x = int(5.0) # int
x = str(5.0) # string '5.0'
str(5.0) == 5.0 # False
int(5.3) # 5

original_set = {1, 2, 3, 4, 5}
converted_tuple = tuple(original_set)

original_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
converted_list = list(original_dict.items())
```

```
float("hello")
```

> ValueError: could not convert string to float: 'hello'

Empties

Sometimes you'll want to create empty objects that will be filled later on.

```
lst = list() # Or:
lst = []
```

There's no real difference between the two methods above, `[]` is apparently marginally faster.

```
tup = tuple() # Or:
tup = ()
```

```
dic = dict() # Or:
dic = {}
```

```
st = set()
```

None

`NoneType` is its own type in Python. It only has one possible value, `None` - it represents an object with no value.

```
x = None
print(x)
```

> None

```
type(x)
```

> NoneType

Control structures

Conditionals (1/2)

Conditional statements allow us to write programs where only certain blocks of code are executed depending on the state of the program. Let's look at some examples and take note of the keywords, syntax and indentation.

```
name = "Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name!")
print("Nice to meet you!")
```

> That's my name too!
>
> Nice to meet you!

Conditionals (2/2)

The main points to notice:

- Use keywords `if`, `elif` and `else`.
- The colon `:` ends each conditional expression.
- Indentation (by 4 empty space) defines code blocks.
- In an `if` statement, the first block whose conditional statement returns `True` is executed and the program exits the `if` block.
- `if` statements don't necessarily need `elif` or `else`.
- `elif` lets us check several conditions.
- `else` lets us evaluate a default block if all other conditions are `False`.
- the end of the entire `if` statement is where the indentation returns to the same level as the first `if` keyword.

Conditionals: nesting

If statements can also be **nested** inside of one another:

```
name = "Super Tom"

if name.lower() == "tom":
    print("That's my name too!")
elif name.lower() == "santa":
    print("That's a funny name.")
else:
    print(f"Hello {name}! That's a cool name.")
    if name.lower().startswith("super"):
        print("Do you really have superpowers?")

print("Nice to meet you!")
```

> Hello Super Tom! That's a cool name.
>
> Do you really have superpowers?
>
> Nice to meet you!

Inline `if`/`else`

We can write simple `if` statements *inline*, i.e., in a single line, for simplicity (similar to the ternary operator `(condition) ? if_true : if_false` in C++).

```
words = ["the", "list", "of", "words"]

x = "long list" if len(words) > 10 else "short list"

# This is equivalent to:
if len(words) > 10:
    x = "long list"
else:
    x = "short list"
```

Truth value testing (1/2)

Any object can be tested for *truth* in Python, for use in `if` and `while` statements.

- `True` values: all objects return `True` unless they are a `bool` object with value `False` or have `len()` == 0
- `False` values: `None`, `False`, `0`, empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`

Read more in the documentation here.

```
x = 1

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

> I'm truthy!

Truth value testing (2/2)

```
x = False

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

> I'm falsey!

```
x = []

if x:
    print("I'm truthy!")
else:
    print("I'm falsey!")
```

> I'm falsey!

`for` loops (1/2)

`for` loops allow us to execute code a specific number of times.

```
for n in [2, 7, -1, 5]:
    print(f"The number is {n} and its square is {n**2}")
print("I'm outside the loop!")
```

> The number is 2 and its square is 4
>
> The number is 7 and its square is 49
>
> The number is -1 and its square is 1
>
> The number is 5 and its square is 25
>
> I'm outside the loop!

`for` loops (2/2)

The main points to notice:

- Keyword `for` begins the loop. Colon `:` ends the first line of the loop.
- Block of code indented is executed for each value in the list (hence the name *for* loops).
- The loop ends after the variable `n` has taken all the values in the list.
- We can iterate over any kind of *iterable*: `range`, `string`, `list`, `tuple`, `set`, `dict`.
- An iterable is really just any object with a sequence of values that can be looped over. In this case, we are iterating over the values in a list.

```
word = "Python"
for letter in word:
    print("Gimme a " + letter + "!")

print(f"What's that spell? {word}!")
```

`range` (1/2)

A very common pattern is to use `for` with the `range()`. `range()` gives you a sequence of integers up to some value (non-inclusive of the end-value) and is typically used for looping.

```
range(10)
```

> range(0, 10)

```
list(range(10))
```

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

`range` (2/2)

```
for i in range(10):
    print(i)
```

We can also specify a start value and a skip-by value with `range`:

```
for i in range(1, 101, 10):
    print(i)
```

> 1
> 11
> 21
> ...
> 91

Nested `for` loops

We can write a loop inside another loop to iterate over multiple dimensions of data:

```
for x in [1, 2, 3]:
    for y in ["a", "b", "c"]:
        print((x, y))
```

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

`zip`

`zip` returns a zip object which is an iterable of tuples.

```
for i in zip(list_1, list_2):
    print(i)
```

> (0, 'a')
>
> (1, 'b')
>
> (2, 'c')

We can even *unpack* these tuples directly in the `for` loop:

```
for i, j in zip(list_1, list_2):
    print(i, j)
```

> 0 a
>
> 1 b
>
> 2 c

`enumerate`

`enumerate` adds a counter to an iterable which we can use within the loop.

```
for i in enumerate(list_2):
    print(i)
```

> (0, 'a')
>
> (1, 'b')
>
> (2, 'c')

```
for n, i in enumerate(list_2):
    print(f"index {n}, value {i}")
```

> index 0, value a
>
> index 1, value b
>
> index 2, value c

Looping over dictionaries

We can loop through key-value pairs of a dictionary using `.items()`. The general syntax is `for key, value in dictionary.items()`.

```
courses = {"Programming": "awesome!",
           "Statistics": "naptime!"}

for course, description in courses.items():
    print(f"{course} is {description}")
```

> Programming is awesome!
>
> Statistics is naptime!

```
for n, (course, description) in enumerate(courses.items()):
    print(f"Item {n}: {course} is {description}")
```

> Item 0: Programming is awesome!
>
> Item 1: Statistics is naptime!

`while` loops

We can also use a `while` loop to excute a block of code until a condition is met.

```
n = 10
while n > 0:
    print(n)
    n -= 1
```

```
10
9
8
...
1
```

`break`

Hence, in some cases, you may want to force a loop to stop based on some criteria, using the `break` keyword.

```
n = 123
i = 0
while n != 1:
    print(int(n))
    if n % 2 == 0: # n is even.
        n = n / 2
    else: # n is odd.
        n = 3 * n + 1
    i += 1
    if i == 10:
        print(f"Ugh, too many iterations!")
        break
```

> ...
> Ugh, too many iterations!

`continue`

The `continue` keyword is similar to `break` but won't stop the loop. Instead, it just restarts the loop from the next iteration.

```
n = 10
while n > 0:
    if n % 2 != 0: # n is odd.
        n = n - 1
        continue
    print(n)
    n = n - 1
```

```
10
8
6
4
2
```

Comprehensions

Comprehensions

Comprehensions allow us to build lists/tuples/sets/dictionaries in one convenient, compact line of code. I use these quite a bit! Below is a standard `for` loop you might use to iterate over an iterable and create a list:

```
subliminal = ['Tom', 'ingests', 'many', 'eggs', 'to', 'outrun', 'large', 'eagles', 'after', 'running', 'near', '!']
first_letters = []
for word in subliminal:
    first_letters.append(word[0])
print(first_letters)
```

> ['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']

List comprehension allows us to do this in one compact line:

```
letters = [word[0] for word in subliminal]  # List comprehension.
letters
```

> ['T', 'i', 'm', 'e', 't', 'o', 'l', 'e', 'a', 'r', 'n', '!']

Multiple comprehensions

We can make things more complicated by doing multiple iteration or conditional iteration:

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

> [(0, 0), (0, 1), (0, 2), ..., (2, 3)]

```
[i for i in range(11) if i % 2 == 0]
```

> [0, 2, 4, 6, 8, 10]

```
[-i if i % 2 else i for i in range(11)]
```

> [0, -1, 2, -3, 4, -5, 6, -7, 8, -9, 10]

Set and dict comprehensions

Set comprehension:

```
words = ['hello', 'goodbye', 'the', 'antidisestablishmentarianism']
y = {word[-1] for word in words} # {'e', 'o', 'm'}
```

Dictionary comprehension:

```
word_lengths = {word:len(word) for word in words}
# {'hello': 5, 'goodbye': 7, 'the': 3, 'antidisestablishmentarianism': 28}
```

Tuple comprehension doesn't work as you might expect: we get a **generator** instead (explained below).

Exceptions

Exceptions

If something goes wrong, we don't want our code to crash - we want it to **fail gracefully**. In Python, this can be accomplished using `try` / `except`. Here is a basic example:

```
this_variable_does_not_exist
print("Another line") # Code fails before getting to this line.
```

> NameError: name 'this_variable_does_not_exist' is not defined

try-except

```
try:
    this_variable_does_not_exist
except:
    pass # Do nothing.
    print("You did something bad! But I won't raise an error.")
print("Another line")
```

```
You did something bad! But I won't raise an error.
Another line
```

Python tries to execute the code in the `try` block. If an error is encountered, we *catch* this in the `except` block (also called `try` / `catch` in other languages). There are many different error types, or **exceptions** - we saw `NameError` above.

```
5 / 0
```

> ZeroDivisionError: division by zero

More exception types

```
my_list = [1, 2, 3]
my_list[5]
```

> IndexError: list index out of range

```
my_tuple = (1,2,3)
my_tuple[0] = 0
```

> TypeError: 'tuple' object does not support item assignment

Raise exceptions

We can also write code that raises an exception on purpose, using `raise`:

```
def add_one(x):
    if not isinstance(x, float) and not isinstance(x, int):
        raise TypeError(f"Sorry, x must be numeric, you entered a {type(x)}.")

    return x + 1

add_one("blah")
```

> TypeError: Sorry, x must be numeric, you entered a <class 'str'>.

This is useful when your function is complicated and would fail in a complicated way, with a weird error message. You can make the cause of the error much clearer to the *user* of the function.

Finally, we can even define our own exception types by inheriting from the `Exception` class - we'll explore classes and inheritance in the next lecture!

Functions

Functions

A function is a reusable piece of code that can accept input parameters, also known as *arguments*. For example, let's define a function called `square` which takes one input parameter `n` and returns the square `n**2`:

```
def square(n):
    n_squared = n**2
    return n_squared
```

```
square(2) # 4
square(100) # 10000
square(12345) # 152399025
```

Functions begin with the `def` keyword, then the function name, arguments in parentheses, and then a colon (`:`). The code executed by the function is defined by indentation. The output or *return* value of the function is specified using the `return` keyword.

Local variables

When you create a variable inside a function, it is local, which means that it only exists inside the function. For example:

```
def cat_string(str1, str2):
    string = str1 + str2
    return string

cat_string('My name is ', 'Tom')
```

> 'My name is Tom'

```
string
```

> NameError: name 'string' is not defined

Mutable vs. immutable input arguments (1/2)

Strings and tuples are immutable types which means they can't be modified. Lists are mutable and we can assign new values for its various entries. This is the main difference between lists and tuples.

```
names_list = ["Indiana", "Fang", "Linsey"]
names_list[0] = "Cool guy" # Ok.

names_tuple = ("Indiana", "Fang", "Linsey")
names_tuple[0] = "Not cool guy"
```

> TypeError: 'tuple' object does not support item assignment

Same goes for strings. Once defined we cannot modifiy the characters of the string.

```
my_name = "Tom"
my_name[-1] = "q"
```

> TypeError: 'str' object does not support item assignment

Mutable vs. immutable input arguments (2/2)

```
x = ([1, 2, 3], 5)
x[1] = 7
```

> TypeError: 'tuple' object does not support item assignment

```
x[0][1] = 4 # Ok. We are modifying a list here.
```

⚠️ Warning

In Python, input arguments are passed by **reference**.

- When **mutable** objects are passed to a function, changes made to the object inside the function affect the original object outside the function.
- When **immutable** objects are passed to a function, they cannot be modified inside the function, hence the original object remains unchanged.

Side effects

If a function changes the variables passed into it, then it is said to have **side effects**. For example:

```
def silly_sum(my_list):
    my_list.append(0)
    return sum(my_list)
```

```
l = [1, 2, 3, 4]
out = silly_sum(l) # 10
```

The above looks like what we wanted? But it changed our `l` object.

```
l
```

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

If your function has side effects like this, you must mention it in the documentation.

Null return type

If you do not specify a return value, the function returns `None` when it terminates:

```
def f(x):
    x + 1 # No return.
    if x == 999:
        return
print(f(0))
```

> None

Default arguments (1 / 2)

Sometimes it is convenient to have *default values* for some arguments in a function. Because they have default values, these arguments are optional, and are hence called *optional arguments*. For example:

```
def repeat_string(s, n = 2):
    return s * n
```

```
repeat_string("abc", 5)
```

> 'abcabcabcabcabc'

```
repeat_string("abc")
```

> 'abcabc'

You can have any number of required arguments and any number of optional arguments.

Default arguments (2 / 2)

All the optional arguments must come after the required arguments. The required arguments are mapped by the order they appear. The optional arguments can be specified out of order when using the function.

```
def example(a, b, c = "DEFAULT", d = "DEFAULT"):
    print(a, b, c, d)

example(1, 2, 3, 4) # 1 2 3 4
```

- Using the defaults for `c` and `d`:

  ```
  example(1, 2) # 1 2 DEFAULT DEFAULT
  ```

- Specifying `c` and `d` as **keyword arguments** (i.e. by name):

  ```
  example(1, 2, c=3, d=4) # 1 2 3 4
  ```

Type hints

Type hinting is exactly what it sounds like, it hints at the data type of function arguments. You can indicate the type of an argument in a function using the syntax `argument : dtype`, and the type of the return value using `def func() -> dtype`. Let's see an example:

```
def repeat_string(s: str, n: int = 2) -> str:
    return s * n
```

Type hinting just helps your users and IDE identify dtypes and possible bugs. It's just another level of documentation. They do not force users to use that date type, for example, I can still pass an `dict` to `repeat_string` if I want to:

```
repeat_string({'key_1': 1, 'key_2': 2})
```

> TypeError: unsupported operand type(s) for *: 'dict' and 'int'

Multiple return values (1/2)

In many programming languages, functions can only return one object. That is technically true in Python too, but there is a *workaround*, which is to return a tuple.

```
def sum_and_product(x, y):
    return (x + y, x * y)

sum_and_product(5, 6) # (11, 30)
```

The parentheses can be omitted (and often are), and a `tuple` is implicitly returned as defined by the use of the comma:

```
def sum_and_product(x, y):
    return x + y, x * y

sum_and_product(5, 6) # (11, 30)
```

Multiple return values (2/2)

It is common to immediately unpack a returned tuple into separate variables, so it really feels like the function is returning multiple values:

```
s, p = sum_and_product(5, 6)
```

As an aside, it is conventional in Python to use `_` for values to be discarded:

```
s, _ = sum_and_product(5, 6)
```

⚠️ **Warning**: `_` becomes an actual variable name!

Unpacking (1/3)

In Python, the asterisk (`*`) is used for unpacking iterable objects. It allows you to extract the elements from an iterable (e.g., a list, tuple) or the key-value pairs from a dictionary.

Here are a few common use cases for the asterisk for unpacking:

1. **Unpacking in function calls:** When calling a function, you can use the asterisk to unpack the elements of a list, tuple, or any iterable as individual arguments to the function.

```
def add_numbers(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
result = add_numbers(*numbers)
print(result) # 6
```

Unpacking (2/3)

2. **Unpacking in iterables:** You can use the asterisk to unpack elements from one iterable into another.

```
first_list = [1, 2, 3]
second_list = [4, 5, 6]
combined_list = [*first_list, *second_list]
print(combined_list) # [1, 2, 3, 4, 5, 6]
```

3. **Unpacking in tuple assignment:** The asterisk can be used in tuple assignment to capture multiple elements at once.

```
first, *rest = [1, 2, 3, 4, 5]
print(first) # 1
print(rest)  # [2, 3, 4, 5]
```

Unpacking (3/3)

4. **Unpacking in dictionary merging:** When merging dictionaries, the double asterisk (`**`) is used to unpack the key-value pairs from one dictionary into another.

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
merged_dict = {**dict1, **dict2}
print(merged_dict) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

5. **Extended unpacking in function definitions:** In function definitions, you can use the asterisk to collect variable positional arguments (`*args`) and variable keyword arguments (`**kwargs`).

```
def example_function(a, b, *args, **kwargs):
    # a and b are regular arguments
    # args is a tuple of positional arguments
    # kwargs is a dictionary of keyword arguments
    pass
```

Functions with arbitrary number of arguments: `*args`

In Python, `*args` and `**kwargs` are used in function definitions to allow a variable number of arguments.

- `*args` (Arbitrary positional arguments): It allows a function to accept a variable number of positional arguments. The `*args` parameter is used to collect any number of positional arguments into a tuple.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args(1, 2, 3, "four")
```

Functions with arbitrary number of arguments: `**kwargs`

- `**kwargs` (Arbitrary keyword arguments): It allows a function to accept a variable number of keyword arguments. The `**kwargs` parameter is used to collect any number of keyword arguments into a dictionary.

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_kwargs(name="John", age=25, city="New York")
```

Combining `*args` and `**kwargs`

You can use both `*args` and `**kwargs` in the same function definition to accept any combination of positional and keyword arguments.

```
def print_args_and_kwargs(*args, **kwargs):
    for arg in args:
        print(arg)
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_args_and_kwargs(1, 2, 3, name="John", age=25)
```

```
1
2
3
name: John
age: 25
```

Functions as a data type

In Python, functions are actually a data type:

```
def do_nothing(x):
    return x

type(do_nothing) # function
```

This means you can pass functions as arguments into other functions.

```
def square(y):
    return y ** 2

def evaluate_function_on_x_plus_1(fun, x):
    return fun(x + 1)

evaluate_function_on_x_plus_1(square, 5) # 36
```

Anonymous functions

There are two ways to define functions in Python. The way we've been using up until now:

```
def add_one(x):
    return x + 1

add_one(7.2) # 8.2
```

Or by using the `lambda` keyword:

```
add_one = lambda x: x + 1
type(add_one) # function
```

The two approaches above are identical. The one with `lambda` is called an **anonymous function**. Anonymous functions can only take up one line of code, so they aren't appropriate in most cases, but can be useful for smaller things.

Generators (1/3)

Recall list comprehension:

```
[n for n in range(10)]
```

Comprehensions evaluate the entire expression at once, and then returns the full data product. Sometimes, we want to work with just one part of our data at a time, for example, when we can't fit all of our data in memory. For this, we can use *generators*.

```
(n for n in range(10))
```

> <generator object at 0x7f06c9b9ba70>

Notice that we just created a `generator object`. Generator objects are like a *recipe* for generating values. They don't actually do any computation until they are asked to.

Generators (2/3)

We can get values from a generator in three main ways:

- Using `next()`
- Using `list()`
- Looping

```
gen = (n for n in range(10))
next(gen) # 0
next(gen) # 1
```

Once the generator is exhausted, it will raise a `StopIteration` exception.

We can see all the values of a generator using `list()` but this defeats the purpose of using a generator in the first place:

```
gen = (n for n in range(10))
list(gen) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Generators (3/3)

Finally, we can iterate over generator objects too:

```
gen = (n for n in range(10))
for i in gen:
    print(i)
```

Above, we saw how to create a generator object using comprehension syntax but with parentheses. We can also create a generator using functions and the `yield` keyword:

```
def gen():
    for n in range(10):
        yield (n, n ** 2)

g = gen()
next(g) # (0, 0)
next(g) # (1, 1)
next(g) # (2, 4)
```

Docstrings

Docstrings

Writing good functions brings up the idea of function documentation, called "docstrings". The docstring goes right after the `def` line and is wrapped in **triple quotes** `"""`.

```
def make_palindrome(string):
    """Turns the string into a palindrome by concatenating itself with a reversed version of itself."""

    return string + string[::-1]
```

In Python we can use the `help()` function to view another function's documentation. In Jupyter, we can use `?` to view the documentation string of any function in our environment.

```
make_palindrome?
```

Docstring: structure

General docstring convention in Python is described in PEP 257 - Docstring Conventions. There are many different docstring style conventions used in Python. The exact style you use can be important for helping you to render your documentation, or for helping your IDE parse your documentation. Common styles include:

1. **Single-line**: If it's short, then just a single line describing the function will do (as above).
2. **reST style**: see here.
3. **NumPy style**: see here.
4. **Google style**: see here.

Docstrings: the NumPy style

```
def function_name(param1, param2, param3):
    """First line is a short description of the function.

    A paragraph describing in a bit more detail what the
    function does and what algorithms it uses and common
    use cases.

    Parameters
    ----------
    param1 : datatype
        A description of param1.
    param2 : datatype
        A description of param2.
    param3 : datatype
        A longer description because maybe this requires
        more explanation and we can use several lines.

    Returns
    -------
    datatype
        A description of the output, datatypes and behaviours.
        Describe special cases and anything the user needs to
        know to use the function.

    Examples
    --------
    >>> function_name(3,8,-5)
    2.0
    """
```

Docstrings: the NumPy style (example)

```
def make_palindrome(string):
    """Turns the string into a palindrome by concatenating
    itself with a reversed version of itself.

    Parameters
    ----------
    string : str
        The string to turn into a palindrome.

    Returns
    -------
    str
        string concatenated with a reversed version of string

    Examples
    --------
    >>> make_palindrome('tom')
    'tommot'
    """
    return string + string[::-1]
```

Docstrings with optional arguments

```
def repeat_string(s, n = 2):
    """
    Repeat the string s, n times.

    Parameters
    ----------
    s : str
        the string
    n : int, optional
        the number of times, by default = 2

    Returns
    -------
    str
        the repeated string

    Examples
    --------
    >>> repeat_string("Blah", 3)
    "BlahBlahBlah"
    """
    return s * n
```

Modules and packages.
Object-oriented programming.
Classes, inheritance and polymorphism.