# Intro to Data Analysis with Python, Session 1

## Why Python?

- versatility
- ease of use
- extensibility
- ubiquity

## Interacting with Python

- interactive Python
- writing/running scripts
- Jupyter notebooks (literate programming)
- building/using applications

## Working with numbers

Python can be used to perform calculations with integers and floating point values. See the examples below:

### addition, subtraction, multiplication

```
>>> 2 + 2
4

>>> 2 - 1.2
0.8

>>> 8 * 0.5 - 1
3.0
```

### division

```
>>> 9 / 2  # normal division
4.5

>>> 9 // 2  # integer division (aka floor division)
4

>>> 9 % 2  # modulo operator (remainder)
1
```

### powers

```
# 2 cubed
>>> 2 ** 3
8

# (1 + 1) ** 3  # parentheses can be used for grouping
8
```

In [ ]:

```
# Exercise: experiment with numeric calculations
# try replacing the expression below with some of your own

7 + 7 * 5
```

Out[ ]:

```
42
```

## Working with text

Python can also be used to manipulate text strings, which are enclosed in either single or double quotes. See the examples below:

### String examples/dealing with quotes in strings

```
>>> "Caesar Salad"
'Caesar Salad'

>>> "Chef's Salad"  # Ok, since single quote is enclosed in double quotes
"Chef's Salad"

>>> 'Chef\'s Salad' # Ok, because quote is "escaped" by backslash
"Chef's Salad"
```

### String operations

```
>>> "foo" + "bar"  # the + operator concatenates strings
'foobar'

>>> "foo" * 3  # multiplying by an integer repeats a string
'foofoofoo'
```

### String indexing

The character at a specific position (starting with zero) can be retrieved with `string[index]`. Use negative index values to count from the end of the string.

```
>>> "foobar"[0]  # returns the first (0th) character of the string
'f'

>>> "foobar"[3]  # returns the fourth character of the string
'b'

>>> "foobar"[-1]  # returns the last character of the string
'r'
```

```
>>> "foobar"[-3]  # returns the third-to-last character of the string
'b'
```

## String slicing

A specific portion, or slice, of a string can be retrieved with `string[start:end]`. Note that the character at the start index will be included, while the character at the end index will not. If either the start or end index is omitted, the slice retrieved will extend to the start or end of the string.

```
>>> "foobar"[0:3]  # returns the first three characters of the string
'foo'
```

```
>>> "foobar"[:3]  # also returns the first three characters of the
string
'foo'
```

```
>>> "foobar"[-3:]  # returns the last three characters of the string
'bar'
```

## String interpolation

When you want to include a numeric value in a string, you can convert it to a string with the built in `str()` function...

```
In [ ]: "A large pizza is " + str(18) + " inches in diameter."
```

```
Out[ ]: 'A large pizza is 18 inches in diameter.'
```

...or you can use an "f-string" to include one or more expressions and convert them to strings automatically.

```
In [ ]: f"Three cubed is {3 ** 3}."
```

```
Out[ ]: 'Three cubed is 27.'
```

```
In [ ]: # Exercise: experiment with string manipulation
        # try replacing the expression below with some of your own

        "karma " * 5 + "chameleon"
```

```
Out[ ]: 'karma karma karma karma karma chameleon'
```

# Variables and data types

## Variable Assignment

Values are assigned to variables using the equal sign ( `=` ) operator. Since Python uses dynamic typing, there is no need to explicitly declare variable types. Variable assignment does not result in any output, but the built-in `print()` function can be used to output a variable's value.

In [ ]:

```
food = "salad"
print(food)
```

```
salad
```

Note that reassigning to an existing variable is possible (and be careful with this).

In [ ]:

```
print(food)
food = "pizza"
print(food)
```

```
salad
pizza
```

## Common data types

Python has many standard data types, but these are the ones to learn first:

- integer
- float
- string
- boolean ( `True` / `False` )
- list
- dictionary

Data types can be retrieved with the built-in `type()` function.

In [ ]:

```
type("pizza")
```

Out[ ]:

```
str
```

In [ ]:

```
type(food)
```

Out[ ]:

```
str
```

## Lists

Lists are ordered collections of objects which can be written as a list of comma-separated values (items) between square brackets. Lists can be indexed and sliced, like strings.

In [ ]:

```
toppings = ["pepperoni", "black olives", "mushrooms", "anchovies"]

print(f"second item in toppings: {toppings[1]}")
print(f"list containing first two items in toppings: {toppings[:2]}")
```

```
second item in toppings: black olives
list containing first two items in toppings: ['pepperoni', 'black olives']
```

Lists have an `append()` method that can be used to add new items. This modifies the existing list in place.

In [ ]:

```
toppings.append("sausage")
print(toppings)
```

```
['pepperoni', 'black olives', 'mushrooms', 'anchovies', 'sausage']
```

Lists can also be concatenated with the `+` operator.

In [ ]:

```
ingredients = toppings + ["flour", "yeast", "salt", "water", "tomatoes"]
print(ingredients)
```

```
['pepperoni', 'black olives', 'mushrooms', 'anchovies', 'sausage', 'flour', 'ye
ast', 'salt', 'water', 'tomatoes']
```

## Dictionaries

Dictionaries are unordered collections of keys and associated values. Dictionaries can be written as a comma-separated set of key:value pairs within braces `{}`. Values can be retrieved using the corresponding key.

In [ ]:

```
pizza_diameters = {
    "small": 10,
    "medium": 14,
    "large": 18
}

medium_diameter = pizza_diameters["medium"]
print(f"The diameter of a medium pizza is {medium_diameter} inches.")
```

```
The diameter of a medium pizza is 14 inches.
```

Dictionaries and lists can also be nested.

In [ ]:

```
pizza_info = {
    "small": {
        "diameter": 10,
        "base_price": 12.50,
        "price_per_topping": 0.50
    },
    "medium": {
        "diameter": 14,
        "base_price": 17.50,
        "price_per_topping": 0.75
    },
    "large": {
        "diameter": 18,
        "base_price": 21.50,
        "price_per_topping": 1.00
    },
    "sicilian": {
        "width": 9,
        "length": 13,
        "base_price": 17.50,
        "price_per_topping": 0.75
    }
}

large_base_price = pizza_info["large"]["base_price"]
large_price_per_topping = pizza_info["large"]["price_per_topping"]
```

```
print(f"A large pizza costs ${large_base_price:.2f} plus",
      f"${large_price_per_topping:.2f} per topping.")
```

```
A large pizza costs $21.50 plus $1.00 per topping.
```

## Comparing values

Comparison operators can be used to compare values of the same type, producing boolean values. The standard comparison operators are the following:

- `<` (less than)
- `>` (greater than)
- `==` (equal to)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `!=` (not equal to)

In [ ]:

```
food == "salad"
```

Out[ ]:

```
False
```

In [ ]:

```
7 * 33 >= 100
```

Out[ ]:

```
True
```

In [ ]:

```
"anchovies" < "pepperoni"   # strings are compared byte-by-byte
```

Out[ ]:

```
True
```

In [ ]:

```
ingredients == toppings
```

Out[ ]:

```
False
```

# Control flow

If statements are used to apply different logic based on specified conditions.

In [ ]:

```
style = "neapolitan"

if style == "sicilian":
    shape = "rectangle"
else:
    shape = "circle"

print(shape)
```

```
circle
```

In [ ]:

```
size = "large"

if size == "small":
    diameter = 10
elif size == "medium":
    diameter = 14
elif size == "large":
    diameter = 18

print(f"The diameter of a {size} pizza is {diameter} inches.")
```

```
The diameter of a large pizza is 18 inches.
```

While loops can be used to perform an action as long as a specific condition is satisfied. In the example below, we print the square of each value from zero to ten.

In [ ]:

```
i = 0
while i <= 10:
    print(i ** 2)
    i = i + 1
```

```
0
1
4
9
16
25
36
49
64
81
100
```

For loops can be used to iterate over lists, strings, and other "iterable" objects.

In [ ]:

```
for ingredient in ingredients:
    print(ingredient)
```

```
pepperoni
black olives
mushrooms
anchovies
sausage
flour
yeast
salt
water
tomatoes
```

`range()` is useful when a fixed number of iterations is needed.

In [ ]:

```
for i in range(11):
    print(i ** 2)
```

```
0
1
4
9
16
25
36
49
64
81
100
```

In [ ]:

```
# Exercise: Using a loop, print each pizza size/type along with its base price.

# write your code here

for size in pizza_info.keys():
    print(f"{size}: ${pizza_info[size]['base_price']:.2f}")
```

```
small: $12.50
medium: $17.50
large: $21.50
sicilian: $17.50
```

# Functions

Functions allow us to abstract a chunk of logic into a "callable" object. Functions take zero or more arguments (specified in parentheses) and optionally return a value.

In [ ]:

```
# Example

def circle_area(diameter):
    pi = 3.14159
    radius = diameter / 2
    area = pi * radius ** 2
    return area

round(circle_area(18), 3)
```

Out[ ]:

```
254.469
```

In [ ]:

```
# Exercise: Referring to the pizza_info dictionary defined above, write a
# function to calculate the cost of a pizza per square inch based on the
# size/type and number of toppings. Use 3.14159 as the value of pi, and
# be sure to round the price to the nearest cent.

def per_sq_inch_cost(type, topping_count):
    if type == "sicilian":
        area = pizza_info["sicilian"]["width"] * pizza_info["sicilian"]["length"
    else:
        area = circle_area(pizza_info[type]["diameter"])
    price = pizza_info[type]["base_price"] + topping_count * pizza_info[type]["p
    return round(price / area, 2)
```

In [ ]:

```
# Exercise check: if your function is working, this cell should output True

round(per_sq_inch_cost("large", 3), 3) == 0.10 and round(
    per_sq_inch_cost("sicilian", 1), 3
) == 0.16
```

Out[ ]:

```
True
```

# Importing modules/packages

Code can be imported from built-in and third-party modules and from other Python files using the `import` command.

In [ ]:

```
# Example: built-in module

import math

math.pi
```

Out[ ]:

```
3.141592653589793
```

In [ ]:

```
# Example: custom module

from shared_data import pizza_utilities

pizza_utilities.calculate_area_from_diameter(18)
```

Out[ ]:

```
254.46900494077323
```

## Classes

Classes provide a way of bundling data and functionality together. Once a class is defined, multiple instances of the class can be created.

In [ ]:

```
# Example

from datetime import datetime


class Pizza:
    restaurant = "Python Pizzeria"  # class variable shared by all instances

    def __init__(self, customer, type, toppings):
        self.customer = customer  # instance variables unique to each instance
        self.type = type
        self.toppings = toppings
        self.price = (
            pizza_info[type]["base_price"]
            + len(toppings) * pizza_info[type]["price_per_topping"]
        )
        self.status = "ordered"
        self.datetime_ordered = datetime.now()
        self.datetime_prepared = None
        self.datetime_delivered = None
        self.current_status_time = self.datetime_ordered

    def prepared(self):
        if self.status == "ordered":
            self.status = "prepared"
            self.datetime_prepared = datetime.now()
            self.current_status_time = self.datetime_prepared
        else:
            raise Exception(
                'Pizza must have a status of "ordered" before it can be prepared
            )

    def delivered(self):
        if self.status == "prepared":
            self.status = "delivered"
            self.datetime_delivered = datetime.now()
```

```
                self.current_status_time = self.datetime_delivered
            else:
                raise Exception(
                    'Pizza must have a status of "prepared" before it can be deliver
                )

    def __str__(self):
        status_string = (
            f"{self.status} at {self.current_status_time.strftime('%I:%M %p')}"
        )
        return (
            f"{self.type} pizza with {', '.join(self.toppings)} for "
            f"{self.customer}: ${self.price:.2f} ({status_string})"
        )
```

In [ ]:

```
orders = []

orders.append(
    Pizza("Drew", "large", ["pepperoni", "mushrooms", "black olives"])
)

p1 = Pizza("Alasdair", "sicilian", ["sausage"])
orders.append(p1)

orders.append(Pizza("Huxley", "large", ["anchovies"]))

orders[0].prepared()
orders[1].prepared()
orders[0].delivered()

print("Python Pizzeria Order List")
for i in range(len(orders)):
    print(f"{i + 1}. {orders[i]}")
```

```
Python Pizzeria Order List
1. large pizza with pepperoni, mushrooms, black olives for Drew: $24.50 (delive
red at 03:06 PM)
2. sicilian pizza with sausage for Alasdair: $18.25 (prepared at 03:06 PM)
3. large pizza with anchovies for Huxley: $22.50 (ordered at 03:06 PM)
```

# Resources

- Python documentation
- PEP8 style guide
- Python Package Index (PyPI)