This course covers the key Python skills you'll need so you can start using Python. The course is ideal for someone with some previous coding experience who wants to add Python to their repertoire.

We'll start with a brief overview of Python syntax, variable assignment, and arithmetic operators.

Hello, Python!

Python was named for the British comedy troupe Monty Python, so we'll make our first Python program a homage to their skit about Spam.

Just for fun, try reading over the code below and predicting what it's going to do when run. (If you have no idea, that's fine!)

Then click the "output" button to see the results of our program.

In [1]:
```
spam_amount = 0
print(spam_amount)

# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings of Spam)
spam_amount = spam_amount + 4

if spam_amount > 0:
    print("But I don't want ANY spam!")

viking_song = "Spam " * spam_amount
print(viking_song)
```

```
0
But I don't want ANY spam!
Spam Spam Spam Spam
```

There's a lot to unpack here! This silly program demonstrates many important aspects of what Python code looks like and how it works. Let's review the code from top to bottom.

In [2]:
```
spam_amount = 0
```

Variable assignment: Here we create a variable called `spam_amount` and assign it the value of 0 using `=`, which is called the assignment operator.

> **Note**: If you've programmed in certain other languages (like Java or C++), you might be noticing some things Python *doesn't* require us to do here:

- we don't need to "declare" `spam_amount` before assigning to it
- we don't need to tell Python what type of value `spam_amount` is going to refer to. In fact, we can even go on to reassign `spam_amount` to refer to a different sort of

thing like a string or a boolean.

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

```
0
```

Function calls:. `print` is a Python function that displays the value passed to it on the screen. We call functions by putting parentheses after their name, and putting the inputs (or *arguments*) to the function in those parentheses.

In [4]:
```
# Ordering Spam, egg, Spam, Spam, bacon and Spam (4 more servings of Spam)
spam_amount = spam_amount + 4
```

The first line above is a **comment**. In Python, comments begin with the `#` symbol.

Next we see an example of reassignment. Reassigning the value of an existing variable looks just the same as creating a variable - it still uses the `=` assignment operator.

In this case, the value we're assigning to `spam_amount` involves some simple arithmetic on its previous value. When it encounters this line, Python evaluates the expression on the right-hand-side of the `=` (0 + 4 = 4), and then assigns that value to the variable on the left-hand-side.

In [5]:
```
if spam_amount > 0:
    print("But I don't want ANY spam!")

viking_song = "Spam Spam Spam"
print(viking_song)
```

```
But I don't want ANY spam!
Spam Spam Spam
```

We won't talk much about "conditionals" until later, but, even if you've never coded before, you can probably guess what this does. Python is prized for its readability and the simplicity.

Note how we indicated which code belongs to the `if`. `"But I don't want ANY spam!"` is only supposed to be printed if `spam_amount` is positive. But the later code (like `print(viking_song)`) should be executed no matter what. How do we (and Python) know that?

The colon (`:`) at the end of the `if` line indicates that a new **code block** is starting. Subsequent lines which are indented are part of that code block.

> **Note**: If you've coded before, you might know that some other languages use `{` curly braces `}` to mark the beginning and end of code blocks. Python's use of meaningful whitespace can be surprising to programmers who are accustomed to other languages, but in practice it can lead to more

> consistent and readable code than languages that do not enforce indentation of code blocks.

The later lines dealing with `viking_song` are not indented with an extra 4 spaces, so they're not a part of the `if` 's code block. We'll see more examples of indented code blocks later when we define functions and using loops.

This code snippet is also our first sighting of a **string** in Python:

`"But I don't want ANY spam!"`
Strings can be marked either by double or single quotation marks. (But because this particular string *contains* a single-quote character, we might confuse Python by trying to surround it with single-quotes, unless we're careful.)

In [6]:
```
viking_song = "Spam " * spam_amount
print(viking_song)
```

```
Spam Spam Spam Spam
```

The `*` operator can be used to multiply two numbers (`3 * 3` evaluates to 9), but we can also multiply a string by a number, to get a version that's been repeated that many times. Python offers a number of cheeky little time-saving tricks like this where operators like `*` and `+` have a different meaning depending on what kind of thing they're applied to. (The technical term for this is operator overloading.)

Numbers and arithmetic in Python

We've already seen an example of a variable containing a number above:

In [7]:
```
spam_amount = 0
```

"Number" is a fine informal name for the kind of thing, but if we wanted to be more technical, we could ask Python how it would describe the type of thing that `spam_amount` is:

In [8]:
```
type(spam_amount)
```

Out[8]:
```
int
```

It's an `int` - short for integer. There's another sort of number we commonly encounter in Python:

In [9]:
```
type(19.95)
```

Out[9]:
```
float
```

A `float` is a number with a decimal place - very useful for representing things like weights or proportions.

`type()` is the second built-in function we've seen (after `print()`), and it's another good one to remember. It's very useful to be able to ask Python "what kind of thing is this?".

A natural thing to want to do with numbers is perform arithmetic. We've seen the `+` operator for addition, and the `*` operator for multiplication. Python also has us covered for the rest of the basic buttons on your calculator:

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	Integer remainder after division of `a` by `b`
`a ** b`	Exponentiation	`a` raised to the power of `b`
`-a`	Negation	The negative of `a`

One interesting observation here is that, whereas your calculator probably just has one button for division, Python can do two kinds. "True division" is basically what your calculator does:

In [10]:
```
print(5 / 2)
print(6 / 2)
```

```
2.5
3.0
```

It always gives us a `float`.

The `//` operator gives us a result that's rounded down to the next integer.

In [11]:
```
print(5 // 2)
print(6 // 2)
```

```
2
3
```

Can you think of where this would be useful? You'll see an example soon in the coding challenges.

Order of operations

The arithmetic we learned in primary school has conventions about the order in which operations are evaluated. Some remember these by a mnemonic such as **PEMDAS** - **P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction.

Python follows similar rules about which calculations to perform first. They're mostly pretty intuitive.

In [12]:
```
8 - 3 + 2
```

Out[12]: 7

In [13]:
```
-3 + 4 * 2
```

Out[13]: 5

Sometimes the default order of operations isn't what we want:

In [14]:
```
hat_height_cm = 25
my_height_cm = 190
# How tall am I, in meters, when wearing my hat?
total_height_meters = hat_height_cm + my_height_cm / 100
print("Height in meters =", total_height_meters, "?")
```

```
Height in meters = 26.9 ?
```

Parentheses are useful here. You can add them to force Python to evaluate sub-expressions in whatever order you want.

In [15]:
```
total_height_meters = (hat_height_cm + my_height_cm) / 100
print("Height in meters =", total_height_meters)
```

```
Height in meters = 2.15
```

Builtin functions for working with numbers

`min` and `max` return the minimum and maximum of their arguments, respectively...

In [16]:
```
print(min(1, 2, 3))
print(max(1, 2, 3))
```

```
1
3
```

`abs` returns the absolute value of an argument:

In [17]:
```
print(abs(32))
print(abs(-32))
```

```
32
32
```

In addition to being the names of Python's two main numerical types, `int` and `float` can also be called as functions which convert their arguments to the corresponding

type:

In [18]:
```
print(float(10))
print(int(3.33))
# They can even be called on strings!
print(int('807') + 1)
```

```
10.0
3
808
```

Your Turn

Now is your chance. Try your **first Python programming exercise**!

You've already seen and used functions such as `print` and `abs`. But Python has many more functions, and defining your own functions is a big part of python programming.

In this lesson, you will learn more about using and defining functions.

Getting Help

You saw the `abs` function in the previous tutorial, but what if you've forgotten what it does?

The `help()` function is possibly the most important Python function you can learn. If you can remember how to use `help()`, you hold the key to understanding most other functions.

Here is an example:

In [1]:
```
help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negati
ve.
```

`help()` displays two things:

1. the header of that function `round(number, ndigits=None)`. In this case, this tells us that `round()` takes an argument we can describe as `number`. Additionally, we can optionally give a separate argument which could be described as `ndigits`.
2. A brief English description of what the function does.

Common pitfall: when you're looking up a function, remember to pass in the name of the function itself, and not the result of calling that function.

What happens if we invoke help on a *call* to the function `round()`? Unhide the output of the cell below to see.

In [2]:
```
help(round(-2.01))
```

Loading [MathJax]/extensions/Safe.js

```
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer litera
l.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __ceil__(...)
 |      Ceiling of an Integral returns itself.
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floor__(...)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(self, format_spec, /)
 |      Default object formatter.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(self, /)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as
an index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
 |  __radd__(self, value, /)
```

Loading [MathJax]/extensions/Safe.js

```
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __round__(...)
|      Rounding an Integral returns itself.
|
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(self, /)
|      Returns size in memory, in bytes.
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
```

Loading [MathJax]/extensions/Safe.js

```
 |      Return self/value.
 |
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  as_integer_ratio(self, /)
 |      Return integer ratio.
 |
 |      Return a pair of integers, whose ratio is exactly equal to the origi
nal int
 |      and with a positive denominator.
 |
 |      >>> (10).as_integer_ratio()
 |      (10, 1)
 |      >>> (-10).as_integer_ratio()
 |      (-10, 1)
 |      >>> (0).as_integer_ratio()
 |      (0, 1)
 |
 |  bit_count(self, /)
 |      Number of ones in the binary representation of the absolute value of
self.
 |
 |      Also known as the population count.
 |
 |      >>> bin(13)
 |      '0b1101'
 |      >>> (13).bit_count()
 |      3
 |
 |  bit_length(self, /)
 |      Number of bits necessary to represent self in binary.
 |
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  to_bytes(self, /, length=1, byteorder='big', *, signed=False)
 |      Return an array of bytes representing an integer.
 |
 |      length
 |        Length of bytes object to use.  An OverflowError is raised if the
 |        integer is not representable with the given number of bytes.  Defa
ult
 |        is length 1.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'bi
g',
 |        the most significant byte is at the beginning of the byte array.
```

Loading [MathJax]/extensions/Safe.js

```
If
 |       byteorder is 'little', the most significant byte is at the end of
the
 |       byte array.  To request the native byte order of the host system,
use
 |       `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |     signed
 |       Determines whether two's complement is used to represent the integ
er.
 |       If signed is False and a negative integer is given, an OverflowErr
or
 |       is raised.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  from_bytes(bytes, byteorder='big', *, signed=False) from builtins.type
 |      Return the integer represented by the given array of bytes.
 |
 |      bytes
 |        Holds the array of bytes to convert.  The argument must either
 |        support the buffer protocol or be an iterable object producing byt
es.
 |        Bytes and bytearray are examples of built-in objects that support
the
 |        buffer protocol.
 |      byteorder
 |        The byte order used to represent the integer.  If byteorder is 'bi
g',
 |        the most significant byte is at the beginning of the byte array.
If
 |        byteorder is 'little', the most significant byte is at the end of
the
 |        byte array.  To request the native byte order of the host system,
use
 |        `sys.byteorder' as the byte order value.  Default is to use 'big'.
 |      signed
 |        Indicates whether two's complement is used to represent the intege
r.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
```

Loading [MathJax]/extensions/Safe.js

```
|  numerator
|      the numerator of a rational number in lowest terms
|
|  real
|      the real part of a complex number
```

Python evaluates an expression like this from the inside out. First it calculates the value of `round(-2.01)`, then it provides help on the output of that expression.

(And it turns out to have a lot to say about integers! After we talk later about objects, methods, and attributes in Python, the help output above will make more sense.)

`round` is a very simple function with a short docstring. `help` shines even more when dealing with more complex, configurable functions like `print`. Don't worry if the following output looks inscrutable... for now, just see if you can pick anything new out from this help.

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

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.
```

If you were looking for it, you might learn that print can take an argument called `sep`, and that this describes what we put between all the other arguments when we print them.

Defining functions

Builtin functions are great, but we can only get so far with them before we need to start defining our own functions. Below is a simple example.

In [4]:
```
def least_difference(a, b, c):
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

Loading [MathJax]/extensions/Safe.js

This creates a function called `least_difference`, which takes three arguments, `a`, `b`, and `c`.

Functions start with a header introduced by the `def` keyword. The indented block of code following the `:` is run when the function is called.

`return` is another keyword uniquely associated with functions. When Python encounters a `return` statement, it exits the function immediately, and passes the value on the right hand side to the calling context.

Is it clear what `least_difference()` does from the source code? If we're not sure, we can always try it out on a few examples:

In [5]:
```
print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7), # Python allows trailing commas in argument l
)
```

```
9 0 1
```

Or maybe the `help()` function can tell us something about it.

In [6]:
```
help(least_difference)
```

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
```

Python isn't smart enough to read my code and turn it into a nice English description. However, when I write a function, I can provide a description in what's called the **docstring**.

Docstrings

In [7]:
```
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    return min(diff1, diff2, diff3)
```

Loading [MathJax]/extensions/Safe.js

The docstring is a triple-quoted string (which may span multiple lines) that comes immediately after the header of a function. When we call `help()` on a function, it shows the docstring.

In [8]:
```
help(least_difference)
```

```
Help on function least_difference in module __main__:

least_difference(a, b, c)
    Return the smallest difference between any two numbers
    among a, b and c.

    >>> least_difference(1, 5, -5)
    4
```

> **Aside:** The last two lines of the docstring are an example function call and result. (The `>>>` is a reference to the command prompt used in Python interactive shells.) Python doesn't run the example call - it's just there for the benefit of the reader. The convention of including 1 or more example calls in a function's docstring is far from universally observed, but it can be very effective at helping someone understand your function. For a real-world example, see this docstring for the numpy function `np.eye`.

Good programmers use docstrings unless they expect to throw away the code soon after it's used (which is rare). So, you should start writing docstrings, too!

Functions that don't return

What would happen if we didn't include the `return` keyword in our function?

In [9]:
```
def least_difference(a, b, c):
    """Return the smallest difference between any two numbers
    among a, b and c.
    """
    diff1 = abs(a - b)
    diff2 = abs(b - c)
    diff3 = abs(a - c)
    min(diff1, diff2, diff3)

print(
    least_difference(1, 10, 100),
    least_difference(1, 10, 10),
    least_difference(5, 6, 7),
)
```

```
None None None
```

Loading [MathJax]/extensions/Safe.js

Python allows us to define such functions. The result of calling them is the special value `None`. (This is similar to the concept of "null" in other languages.)

Without a `return` statement, `least_difference` is completely pointless, but a function with side effects may do something useful without returning anything. We've already seen two examples of this: `print()` and `help()` don't return anything. We only call them for their side effects (putting some text on the screen). Other examples of useful side effects include writing to a file, or modifying an input.

In [10]:
```
mystery = print()
print(mystery)
```

```
None
```

Default arguments

When we called `help(print)`, we saw that the `print` function has several optional arguments. For example, we can specify a value for `sep` to put some special string in between our printed arguments:

In [11]:
```
print(1, 2, 3, sep=' < ')
```

```
1 < 2 < 3
```

But if we don't specify a value, `sep` is treated as having a default value of `' '` (a single space).

In [12]:
```
print(1, 2, 3)
```

```
1 2 3
```

Adding optional arguments with default values to the functions we define turns out to be pretty easy:

In [13]:
```
def greet(who="Colin"):
    print("Hello,", who)

greet()
greet(who="Kaggle")
# (In this case, we don't need to specify the name of the argument, because
greet("world")
```

```
Hello, Colin
Hello, Kaggle
Hello, world
```

Functions Applied to Functions

Here's something that's powerful, though it can feel very abstract at first. You can supply functions as arguments to other functions. Some example may make this clearer:

Loading [MathJax]/extensions/Safe.js

In [14]:
```
def mult_by_five(x):
    return 5 * x

def call(fn, arg):
    """Call fn on arg"""
    return fn(arg)

def squared_call(fn, arg):
    """Call fn on the result of calling fn on arg"""
    return fn(fn(arg))

print(
    call(mult_by_five, 1),
    squared_call(mult_by_five, 1),
    sep='\n', # '\n' is the newline character - it starts a new line
)
```

```
5
25
```

Functions that operate on other functions are called "higher-order functions." You probably won't write your own for a little while. But there are higher-order functions built into Python that you might find useful to call.

Here's an interesting example using the `max` function.

By default, `max` returns the largest of its arguments. But if we pass in a function using the optional `key` argument, it returns the argument `x` that maximizes `key(x)` (aka the 'argmax').

In [15]:
```
def mod_5(x):
    """Return the remainder of x after dividing by 5"""
    return x % 5

print(
    'Which number is biggest?',
    max(100, 51, 14),
    'Which number is the biggest modulo 5?',
    max(100, 51, 14, key=mod_5),
    sep='\n',
)
```

```
Which number is biggest?
100
Which number is the biggest modulo 5?
14
```

Your Turn

Functions open up a whole new world in Python programming. **Try using them yourself**.

Loading [MathJax]/extensions/Safe.js

Booleans

Python has a type of variable called `bool`. It has two possible values: `True` and `False`.

In [1]:
```
x = True
print(x)
print(type(x))
```

```
True
<class 'bool'>
```

Rather than putting `True` or `False` directly in our code, we usually get boolean values from **boolean operators**. These are operators that answer yes/no questions. We'll go through some of these operators below.

Comparison Operations

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

In [2]:
```
def can_run_for_president(age):
    """Can someone of the given age run for president in the US?"""
    # The US Constitution says you must be at least 35 years old
    return age >= 35

print("Can a 19-year-old run for president?", can_run_for_president(19))
print("Can a 45-year-old run for president?", can_run_for_president(45))
```

```
Can a 19-year-old run for president? False
Can a 45-year-old run for president? True
```

Comparisons frequently work like you'd hope

In [3]:
```
3.0 == 3
```

Out[3]: True

But sometimes they can be tricky

In [4]:
```
'3' == 3
```

Out[4]: False

Comparison operators can be combined with the arithmetic operators we've already seen to express a virtually limitless range of mathematical tests. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

In [5]:
```
def is_odd(n):
    return (n % 2) == 1

print("Is 100 odd?", is_odd(100))
print("Is -1 odd?", is_odd(-1))
```

```
Is 100 odd? False
Is -1 odd? True
```

Remember to use `==` instead of `=` when making comparisons. If you write `n == 2` you are asking about the value of n. When you write `n = 2` you are changing the value of n.

Combining Boolean Values

You can combine boolean values using the standard concepts of "and", "or", and "not". In fact, the words to do this are: `and`, `or`, and `not`.

With these, we can make our `can_run_for_president` function more accurate.

In [6]:
```
def can_run_for_president(age, is_natural_born_citizen):
    """Can someone of the given age and citizenship status run for president
    # The US Constitution says you must be a natural born citizen *and* at l
    return is_natural_born_citizen and (age >= 35)

print(can_run_for_president(19, True))
print(can_run_for_president(55, False))
print(can_run_for_president(55, True))
```

```
False
False
True
```

Quick, can you guess the value of this expression?

In [7]:
```
True or True and False
```

Out[7]:
```
True
```

(Click the "output" button to see the answer)

To answer this, you'd need to figure out the order of operations.

For example, `and` is evaluated before `or`. That's why the first expression above is `True`. If we evaluated it from left to right, we would have calculated `True or True`

first (which is `True`), and then taken the `and` of that result with `False`, giving a final value of `False`.

You could try to memorize the order of precedence, but a safer bet is to just use liberal parentheses. Not only does this help prevent bugs, it makes your intentions clearer to anyone who reads your code.

For example, consider the following expression:

```
prepared_for_weather = have_umbrella or rain_level < 5 and
have_hood or not rain_level > 0 and is_workday
```
I'm trying to say that I'm safe from today's weather....

- if I have an umbrella...
- or if the rain isn't too heavy and I have a hood...
- otherwise, I'm still fine unless it's raining *and* it's a workday

But not only is my Python code hard to read, it has a bug. We can address both problems by adding some parentheses:

```
prepared_for_weather = have_umbrella or (rain_level < 5 and
have_hood) or not (rain_level > 0 and is_workday)
```
You can add even more parentheses if you think it helps readability:

```
prepared_for_weather = have_umbrella or ((rain_level < 5) and
have_hood) or (not (rain_level > 0 and is_workday))
```
We can also split it over multiple lines to emphasize the 3-part structure described above:

```
prepared_for_weather = (
    have_umbrella
    or ((rain_level < 5) and have_hood)
    or (not (rain_level > 0 and is_workday))
)
```

Conditionals

Booleans are most useful when combined with *conditional statements*, using the keywords `if`, `elif`, and `else`.

Conditional statements, often referred to as *if-then* statements, let you control what pieces of code are run based on the value of some Boolean condition. Here's an example:

In [8]:
```
def inspect(x):
    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...")

inspect(0)
inspect(-15)
```

```
0 is zero
-15 is negative
```

The `if` and `else` keywords are often used in other languages; its more unique keyword is `elif`, a contraction of "else if". In these conditional clauses, `elif` and `else` blocks are optional; additionally, you can include as many `elif` statements as you would like.

Note especially the use of colons (`:`) and whitespace to denote separate blocks of code. This is similar to what happens when we define a function - the function header ends with `:`, and the following line is indented with 4 spaces. All subsequent indented lines belong to the body of the function, until we encounter an unindented line, ending the function definition.

In [9]:
```
def f(x):
    if x > 0:
        print("Only printed when x is positive; x =", x)
        print("Also only printed when x is positive; x =", x)
    print("Always printed, regardless of x's value; x =", x)

f(1)
f(0)
```

```
Only printed when x is positive; x = 1
Also only printed when x is positive; x = 1
Always printed, regardless of x's value; x = 1
Always printed, regardless of x's value; x = 0
```

Boolean conversion

We've seen `int()`, which turns things into ints, and `float()`, which turns things into floats, so you might not be surprised to hear that Python has a `bool()` function which turns things into bools.

In [10]:
```
print(bool(1)) # all numbers are treated as true, except 0
print(bool(0))
print(bool("asf")) # all strings are treated as true, except the empty strin
print(bool(""))
# Generally empty sequences (strings, lists, and other types we've yet to se
# are "falsey" and the rest are "truthy"
```

```
True
False
True
False
```

We can use non-boolean objects in `if` conditions and other places where a boolean would be expected. Python will implicitly treat them as their corresponding boolean value:

In [11]:
```
if 0:
    print(0)
elif "spam":
    print("spam")
```

```
spam
```

Your Turn

You probably don't realize how much you have learned so far. Go try the **hands-on coding problems**, and you'll be pleasantly surprised about how much you can do.

Lists

Lists in Python represent ordered sequences of values. Here is an example of how to create them:

In [1]:
```
primes = [2, 3, 5, 7]
```

We can put other types of things in lists:

In [2]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
```

We can even make a list of lists:

In [3]:
```
hands = [
    ['J', 'Q', 'K'],
    ['2', '2', '2'],
    ['6', 'A', 'K'], # (Comma after the last element is optional)
]
# (I could also have written this on one line, but it can get hard to read)
hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]
```

A list can contain a mix of different types of variables:

In [4]:
```
my_favourite_things = [32, 'raindrops on roses', help]
# (Yes, Python's help function is *definitely* one of my favourite things)
```

Indexing

You can access individual list elements with square brackets.

Which planet is closest to the sun? Python uses *zero-based* indexing, so the first element has index 0.

In [5]:
```
planets[0]
```

Out[5]:
```
'Mercury'
```

What's the next closest planet?

In [6]:
```
planets[1]
```

Out[6]:
```
'Venus'
```

Which planet is *furthest* from the sun?

Loading [MathJax]/extensions/Safe.js

The end of the list can be accessed with negative numbers, starting from -1:

In [7]:
```
planets[-1]
```

Out[7]:
```
'Neptune'
```

In [8]:
```
planets[-2]
```

Out[8]:
```
'Uranus'
```

Slicing

What are the first three planets? We can answer this question using *slicing*:

In [9]:
```
planets[0:3]
```

Out[9]:
```
['Mercury', 'Venus', 'Earth']
```

`planets[0:3]` is our way of asking for the elements of `planets` starting from index 0 and continuing up to *but not including* index 3.

The starting and ending indices are both optional. If I leave out the start index, it's assumed to be 0. So I could rewrite the expression above as:

In [10]:
```
planets[:3]
```

Out[10]:
```
['Mercury', 'Venus', 'Earth']
```

If I leave out the end index, it's assumed to be the length of the list.

In [11]:
```
planets[3:]
```

Out[11]:
```
['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

i.e. the expression above means "give me all the planets from index 3 onward".

We can also use negative indices when slicing:

In [12]:
```
# All the planets except the first and last
planets[1:-1]
```

Out[12]:
```
['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']
```

In [13]:
```
# The last 3 planets
planets[-3:]
```

Out[13]:
```
['Saturn', 'Uranus', 'Neptune']
```

Changing lists

Loading [MathJax]/extensions/Safe.js

Lists are "mutable", meaning they can be modified "in place".

One way to modify a list is to assign to an index or slice expression.

For example, let's say we want to rename Mars:

In [14]:
```
planets[3] = 'Malacandra'
planets
```

Out[14]:
```
['Mercury',
 'Venus',
 'Earth',
 'Malacandra',
 'Jupiter',
 'Saturn',
 'Uranus',
 'Neptune']
```

Hm, that's quite a mouthful. Let's compensate by shortening the names of the first 3 planets.

In [15]:
```
planets[:3] = ['Mur', 'Vee', 'Ur']
print(planets)
# That was silly. Let's give them back their old names
planets[:4] = ['Mercury', 'Venus', 'Earth', 'Mars',]
```

```
['Mur', 'Vee', 'Ur', 'Malacandra', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

List functions

Python has several useful functions for working with lists.

`len` gives the length of a list:

In [16]:
```
# How many planets are there?
len(planets)
```

Out[16]:
```
8
```

`sorted` returns a sorted version of a list:

In [17]:
```
# The planets sorted in alphabetical order
sorted(planets)
```

Out[17]:
```
['Earth', 'Jupiter', 'Mars', 'Mercury', 'Neptune', 'Saturn', 'Uranus', 'Ven
us']
```

`sum` does what you might expect:

In [18]:
```
primes = [2, 3, 5, 7]
sum(primes)
```

Loading [MathJax]/extensions/Safe.js

Out[18]: 17

We've previously used the `min` and `max` to get the minimum or maximum of several arguments. But we can also pass in a single list argument.

In [19]:
```
max(primes)
```

Out[19]: 7

Interlude: objects

I've used the term 'object' a lot so far - you may have even read that *everything* in Python is an object. What does that mean?

In short, objects carry some things around with them. You access that stuff using Python's dot syntax.

For example, numbers in Python carry around an associated variable called `imag` representing their imaginary part. (You'll probably never need to use this unless you're doing some very weird math.)

In [20]:
```
x = 12
# x is a real number, so its imaginary part is 0.
print(x.imag)
# Here's how to make a complex number, in case you've ever been curious:
c = 12 + 3j
print(c.imag)
```

```
0
3.0
```

The things an object carries around can also include functions. A function attached to an object is called a **method**. (Non-function things attached to an object, such as `imag`, are called *attributes*).

For example, numbers have a method called `bit_length`. Again, we access it using dot syntax:

In [21]:
```
x.bit_length
```

Out[21]: <function int.bit_length()>

To actually call it, we add parentheses:

In [22]:
```
x.bit_length()
```

Out[22]: 4

Loading [MathJax]/extensions/Safe.js

> **Aside:** You've actually been calling methods already if you've been doing the exercises. In the exercise notebooks `q1`, `q2`, `q3`, etc. are all objects which have methods called `check`, `hint`, and `solution`.

In the same way that we can pass functions to the `help` function (e.g. `help(max)`), we can also pass in methods:

In [23]:
```
help(x.bit_length)
```

```
Help on built-in function bit_length:

bit_length() method of builtins.int instance
    Number of bits necessary to represent self in binary.

    >>> bin(37)
    '0b100101'
    >>> (37).bit_length()
    6
```

The examples above were utterly obscure. None of the types of objects we've looked at so far (numbers, functions, booleans) have attributes or methods you're likely ever to use.

But it turns out that lists have several methods which you'll use all the time.

List methods

`list.append` modifies a list by adding an item to the end:

In [24]:
```
# Pluto is a planet darn it!
planets.append('Pluto')
```

Why does the cell above have no output? Let's check the documentation by calling `help(planets.append)`.

> **Aside:** `append` is a method carried around by *all* objects of type list, not just `planets`, so we also could have called `help(list.append)`. However, if we try to call `help(append)`, Python will complain that no variable exists called "append". The "append" name only exists within lists - it doesn't exist as a standalone name like builtin functions such as `max` or `len`.

In [25]:
```
help(planets.append)
```

Loading [MathJax]/extensions/Safe.js

```
Help on built-in function append:

append(object, /) method of builtins.list instance
    Append object to the end of the list.
```

The `-> None` part is telling us that `list.append` doesn't return anything. But if we check the value of `planets`, we can see that the method call modified the value of `planets`:

In [26]:
```
planets
```

Out[26]:
```
['Mercury',
 'Venus',
 'Earth',
 'Mars',
 'Jupiter',
 'Saturn',
 'Uranus',
 'Neptune',
 'Pluto']
```

`list.pop` removes and returns the last element of a list:

In [27]:
```
planets.pop()
```

Out[27]:
```
'Pluto'
```

In [28]:
```
planets
```

Out[28]:
```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

Searching lists

Where does Earth fall in the order of planets? We can get its index using the `list.index` method.

In [29]:
```
planets.index('Earth')
```

Out[29]:
```
2
```

It comes third (i.e. at index 2 - 0 indexing!).

At what index does Pluto occur?

In [30]:
```
planets.index('Pluto')
```

Loading [MathJax]/extensions/Safe.js

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[30], line 1
----> 1 planets.index('Pluto')

ValueError: 'Pluto' is not in list
```

Oh, that's right...

To avoid unpleasant surprises like this, we can use the `in` operator to determine whether a list contains a particular value:

In [31]:
```
# Is Earth a planet?
"Earth" in planets
```

Out[31]:
```
True
```

In [32]:
```
# Is Calbefraques a planet?
"Calbefraques" in planets
```

Out[32]:
```
False
```

There are a few more interesting list methods we haven't covered. If you want to learn about all the methods and attributes attached to a particular object, we can call `help()` on the object itself. For example, `help(planets)` will tell us about *all* the list methods:

In [33]:
```
help(planets)
```

Loading [MathJax]/extensions/Safe.js

```
Help on list object:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the list.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(self, /)
 |      Return the size of the list in memory, in bytes.
 |
 |  append(self, object, /)
 |      Append object to the end of the list.
 |
 |  clear(self, /)
 |      Remove all items from list.
 |
 |  copy(self, /)
 |      Return a shallow copy of the list.
 |
 |  count(self, value, /)
 |      Return number of occurrences of value.
 |
 |  extend(self, iterable, /)
 |      Extend list by appending elements from the iterable.
 |
 |  index(self, value, start=0, stop=9223372036854775807, /)
 |      Return first index of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  insert(self, index, object, /)
 |      Insert object before index.
 |
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
```

Loading [MathJax]/extensions/Safe.js

```
 |      Reverse *IN PLACE*.
 |
 |  sort(self, /, *, key=None, reverse=False)
 |      Sort the list in ascending order and return None.
 |
 |      The sort is in-place (i.e. the list itself is modified) and stable
(i.e. the
 |      order of two equal elements is maintained).
 |
 |      If a key function is given, apply it once to each list item and sort
them,
 |      ascending or descending, according to their function values.
 |
 |      The reverse flag can be set to sort in descending order.
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Click the "output" button to see the full help page. Lists have lots of methods with weird-looking names like `__eq__` and `__iadd__`. Don't worry too much about these for now. (You'll probably never call such methods directly. But they get called behind the scenes when we use syntax like indexing or comparison operators.) The most interesting methods are toward the bottom of the list (`append`, `clear`, `copy`, etc.).

Tuples

Tuples are almost exactly the same as lists. They differ in just two ways.

1: The syntax for creating them uses parentheses instead of square brackets

In [34]:
```
t = (1, 2, 3)
```

In [35]:
```
t = 1, 2, 3 # equivalent to above
t
```

Loading [MathJax]/extensions/Safe.js

2: They cannot be modified (they are *immutable*).

In [36]:
```
t[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 1 t[0] = 100

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

Tuples are often used for functions that have multiple return values.

For example, the `as_integer_ratio()` method of float objects returns a numerator and a denominator in the form of a tuple:

In [37]:
```
x = 0.125
x.as_integer_ratio()
```

Out[37]: (1, 8)

These multiple return values can be individually assigned as follows:

In [38]:
```
numerator, denominator = x.as_integer_ratio()
print(numerator / denominator)
```

```
0.125
```

Finally we have some insight into the classic Stupid Python Trick™ for swapping two variables!

In [39]:
```
a = 1
b = 0
a, b = b, a
print(a, b)
```

```
0 1
```

Your Turn

You learn best by writing code, not just reading it. So try **the coding challenge** now.

Loading [MathJax]/extensions/Safe.js

Loops

Loops are a way to repeatedly execute some code. Here's an example:

In [1]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
for planet in planets:
    print(planet, end=' ') # print all on same line
```

```
Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune
```

The `for` loop specifies

- the variable name to use (in this case, `planet`)
- the set of values to loop over (in this case, `planets`)

You use the word "`in`" to link them together.

The object to the right of the "`in`" can be any object that supports iteration. Basically, if it can be thought of as a group of things, you can probably loop over it. In addition to lists, we can iterate over the elements of a tuple:

In [2]:
```
multiplicands = (2, 2, 2, 3, 3, 5)
product = 1
for mult in multiplicands:
    product = product * mult
product
```

Out[2]:
```
360
```

You can even loop through each character in a string:

In [3]:
```
s = 'steganograpHy is the practicE of conceaLing a file, message, image, or
msg = ''
# print all the uppercase letters in s, one at a time
for char in s:
    if char.isupper():
        print(char, end='')
```

```
HELLO
```

range()

`range()` is a function that returns a sequence of numbers. It turns out to be very useful for writing loops.

For example, if we want to repeat some action 5 times:

```
ange(5):
```

Loading [MathJax]/extensions/Safe.js

```
    print("Doing important work. i =", i)
```

```
Doing important work. i = 0
Doing important work. i = 1
Doing important work. i = 2
Doing important work. i = 3
Doing important work. i = 4
```

`while` loops

The other type of loop in Python is a `while` loop, which iterates until some condition is met:

In [5]:
```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1 # increase the value of i by 1
```

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

The argument of the `while` loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

List comprehensions

List comprehensions are one of Python's most beloved and unique features. The easiest way to understand them is probably to just look at a few examples:

In [6]:
```
squares = [n**2 for n in range(10)]
squares
```

Out[6]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here's how we would do the same thing without a list comprehension:

In [7]:
```
squares = []
for n in range(10):
    squares.append(n**2)
squares
```

Out[7]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can also add an `if` condition:

In [8]:
```
short_planets = [planet for planet in planets if len(planet) < 6]
short_planets
```

Out[8]:
```
['Venus', 'Earth', 'Mars']
```

Loading [MathJax]/extensions/Safe.js

(If you're familiar with SQL, you might think of this as being like a "WHERE" clause)

Here's an example of filtering with an `if` condition *and* applying some transformation to the loop variable:

In [9]:
```
# str.upper() returns an all-caps version of a string
loud_short_planets = [planet.upper() + '!' for planet in planets if len(plar
loud_short_planets
```

Out[9]:
```
['VENUS!', 'EARTH!', 'MARS!']
```

People usually write these on a single line, but you might find the structure clearer when it's split up over 3 lines:

In [10]:
```
[
    planet.upper() + '!'
    for planet in planets
    if len(planet) < 6
]
```

Out[10]:
```
['VENUS!', 'EARTH!', 'MARS!']
```

(Continuing the SQL analogy, you could think of these three lines as SELECT, FROM, and WHERE)

The expression on the left doesn't technically have to involve the loop variable (though it'd be pretty unusual for it not to). What do you think the expression below will evaluate to? Press the 'output' button to check.

In [11]:
```
[32 for planet in planets]
```

Out[11]:
```
[32, 32, 32, 32, 32, 32, 32, 32]
```

List comprehensions combined with functions like `min`, `max`, and `sum` can lead to impressive one-line solutions for problems that would otherwise require several lines of code.

For example, compare the following two cells of code that do the same thing.

In [12]:
```
def count_negatives(nums):
    """Return the number of negative numbers in the given list.

    >>> count_negatives([5, -1, -2, 0, 3])
    2
    """
    n_negative = 0
    for num in nums:
        if num < 0:
            n_negative = n_negative + 1
    n_negative
```

Loading [MathJax]/extensions/Safe.js

Here's a solution using a list comprehension:

In [13]:
```
def count_negatives(nums):
    return len([num for num in nums if num < 0])
```

Much better, right?

Well if all we care about is minimizing the length of our code, this third solution is better still!

In [14]:
```
def count_negatives(nums):
    # Reminder: in the "booleans and conditionals" exercises, we learned abo
    # Python where it calculates something like True + True + False + True t
    return sum([num < 0 for num in nums])
```

Which of these solutions is the "best" is entirely subjective. Solving a problem with less code is always nice, but it's worth keeping in mind the following lines from The Zen of Python:

> Readability counts.
> Explicit is better than implicit.

So, use these tools to make compact readable programs. But when you have to choose, favor code that is easy for others to understand.

Your Turn

You know what's next -- we have some **fun coding challenges** for you! This next set of coding problems is shorter, so try it now.

Loading [MathJax]/extensions/Safe.js

This lesson will cover two essential Python types: **strings** and **dictionaries**.

Strings

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations.

Such string manipulation patterns come up often in the context of data science work.

String syntax

You've already seen plenty of strings in examples during the previous lessons, but just to recap, strings in Python can be defined using either single or double quotations. They are functionally equivalent.

In [1]:
```
x = 'Pluto is a planet'
y = "Pluto is a planet"
x == y
```

Out[1]:
```
True
```

Double quotes are convenient if your string contains a single quote character (e.g. representing an apostrophe).

Similarly, it's easy to create a string that contains double-quotes if you wrap it in single quotes:

In [2]:
```
print("Pluto's a planet!")
print('My dog is named "Pluto"')
```

```
Pluto's a planet!
My dog is named "Pluto"
```

If we try to put a single quote character inside a single-quoted string, Python gets confused:

In [3]:
```
'Pluto's a planet!'
```

```
  Cell In[3], line 1
    'Pluto's a planet!'
           ^
SyntaxError: unterminated string literal (detected at line 1)
```

We can fix this by "escaping" the single quote with a backslash.

In [4]:
```
'Pluto\'s a planet!'
```

Loading [MathJax]/extensions/Safe.js

Out[4]: `"Pluto's a planet!"`

The table below summarizes some important uses of the backslash character.

What you type...	What you get	example	`print(example)`
`\'`	`'`	`'What\'s up?'`	What's up?
`\"`	`"`	`"That's \"cool\""`	That's "cool"
`\\`	`\`	`"Look, a mountain: /\\"`	Look, a mountain: /\
`\n`	`"1\n2 3"`	`1` `2 3`

The last sequence, `\n`, represents the *newline character*. It causes Python to start a new line.

In [5]:
```
hello = "hello\nworld"
print(hello)
```

```
hello
world
```

In addition, Python's triple quote syntax for strings lets us include newlines literally (i.e. by just hitting 'Enter' on our keyboard, rather than using the special '\n' sequence). We've already seen this in the docstrings we use to document our functions, but we can use them anywhere we want to define a string.

In [6]:
```
triplequoted_hello = """hello
world"""
print(triplequoted_hello)
triplequoted_hello == hello
```

```
hello
world
```

Out[6]: `True`

The `print()` function automatically adds a newline character unless we specify a value for the keyword argument `end` other than the default value of `'\n'`:

In [7]:
```
print("hello")
print("world")
print("hello", end='')
print("pluto", end='')
```

```
hello
world
hellopluto
```

Strings are sequences

Loading [MathJax]/extensions/Safe.js

Strings can be thought of as sequences of characters. Almost everything we've seen that we can do to a list, we can also do to a string.

In [8]:
```
# Indexing
planet = 'Pluto'
planet[0]
```

Out[8]:
```
'P'
```

In [9]:
```
# Slicing
planet[-3:]
```

Out[9]:
```
'uto'
```

In [10]:
```
# How long is this string?
len(planet)
```

Out[10]:
```
5
```

In [11]:
```
# Yes, we can even loop over them
[char+'! ' for char in planet]
```

Out[11]:
```
['P! ', 'l! ', 'u! ', 't! ', 'o! ']
```

But a major way in which they differ from lists is that they are *immutable*. We can't modify them.

In [12]:
```
planet[0] = 'B'
# planet.append doesn't work either
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 planet[0] = 'B'
      2 # planet.append doesn't work either

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

String methods

Like `list`, the type `str` has lots of very useful methods. I'll show just a few examples here.

In [13]:
```
# ALL CAPS
claim = "Pluto is a planet!"
claim.upper()
```

Out[13]:
```
'PLUTO IS A PLANET!'
```

In [14]:
```
# all lowercase
claim.lower()
```

Loading [MathJax]/extensions/Safe.js

Out[14]:
```
'pluto is a planet!'
```

In [15]:
```
# Searching for the first index of a substring
claim.index('plan')
```

Out[15]:
```
11
```

In [16]:
```
claim.startswith(planet)
```

Out[16]:
```
True
```

In [17]:
```
# false because of missing exclamation mark
claim.endswith('planet')
```

Out[17]:
```
False
```

Going between strings and lists: `.split()` and `.join()`

`str.split()` turns a string into a list of smaller strings, breaking on whitespace by default. This is super useful for taking you from one big string to a list of words.

In [18]:
```
words = claim.split()
words
```

Out[18]:
```
['Pluto', 'is', 'a', 'planet!']
```

Occasionally you'll want to split on something other than whitespace:

In [19]:
```
datestr = '1956-01-31'
year, month, day = datestr.split('-')
```

`str.join()` takes us in the other direction, sewing a list of strings up into one long string, using the string it was called on as a separator.

In [20]:
```
'/'.join([month, day, year])
```

Out[20]:
```
'01/31/1956'
```

In [21]:
```
# Yes, we can put unicode characters right in our string literals :)
' 👋 '.join([word.upper() for word in words])
```

Out[21]:
```
'PLUTO 👋 IS 👋 A 👋 PLANET!'
```

Building strings with `.format()`

Python lets us concatenate strings with the `+` operator.

In [22]:
```
planet + ', we miss you.'
```

Loading [MathJax]/extensions/Safe.js

Out[22]:
```
'Pluto, we miss you.'
```

If we want to throw in any non-string objects, we have to be careful to call `str()` on them first

In [23]:
```
position = 9
planet + ", you'll always be the " + position + "th planet to me."
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 2
      1 position = 9
----> 2 planet + ", you'll always be the " + position + "th planet to me."

TypeError: can only concatenate str (not "int") to str
```

In [24]:
```
planet + ", you'll always be the " + str(position) + "th planet to me."
```

Out[24]:
```
"Pluto, you'll always be the 9th planet to me."
```

This is getting hard to read and annoying to type. `str.format()` to the rescue.

In [25]:
```
"{}, you'll always be the {}th planet to me.".format(planet, position)
```

Out[25]:
```
"Pluto, you'll always be the 9th planet to me."
```

So much cleaner! We call `.format()` on a "format string", where the Python values we want to insert are represented with `{}` placeholders.

Notice how we didn't even have to call `str()` to convert `position` from an int. `format()` takes care of that for us.

If that was all that `format()` did, it would still be incredibly useful. But as it turns out, it can do a *lot* more. Here's just a taste:

In [26]:
```
pluto_mass = 1.303 * 10**22
earth_mass = 5.9722 * 10**24
population = 52910390
#         2 decimal points   3 decimal points, format as percent     separat
"{} weighs about {:.2} kilograms ({:.3%} of Earth's mass). It is home to {:,
    planet, pluto_mass, pluto_mass / earth_mass, population,
)
```

Out[26]:
```
"Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home
to 52,910,390 Plutonians."
```

In [27]:
```
# Referring to format() arguments by index, starting from 0
s = """Pluto's a {0}.
No, it's a {1}.
{0}!
{1}!""".format('planet', 'dwarf planet')
```

Loading [MathJax]/extensions/Safe.js

```
Pluto's a planet.
No, it's a dwarf planet.
planet!
dwarf planet!
```

You could probably write a short book just on `str.format`, so I'll stop here, and point you to pyformat.info and the official docs for further reading.

Building strings with f-string

This is an alternative modernized way of building strings came with Python 3.6+

In [28]:
```
f"{planet}, you'll always be the {position}th planet to me."
```

Out[28]:
```
"Pluto, you'll always be the 9th planet to me."
```

Comparing to `.format()` command, f-string is cleaner. Especially, to do something like this

In [29]:
```
pluto_mass = 1.303 * 10**22
earth_mass = 5.9722 * 10**24
population = 52910390
pluto_earth_ratio = pluto_mass / earth_mass

f"{planet} weighs about {pluto_mass:.2} kilograms ({pluto_earth_ratio:.3%} c
```

Out[29]:
```
"Pluto weighs about 1.3e+22 kilograms (0.218% of Earth's mass). It is home
to 52,910,390 Plutonians."
```

Dictionaries

Dictionaries are a built-in Python data structure for mapping keys to values.

In [30]:
```
numbers = {'one':1, 'two':2, 'three':3}
```

In this case `'one'`, `'two'`, and `'three'` are the **keys**, and 1, 2 and 3 are their corresponding values.

Values are accessed via square bracket syntax similar to indexing into lists and strings.

In [31]:
```
numbers['one']
```

Out[31]:
```
1
```

We can use the same syntax to add another key, value pair

In [32]:
```
numbers['eleven'] = 11
numbers
```

Loading [MathJax]/extensions/Safe.js

Out[32]:
```
{'one': 1, 'two': 2, 'three': 3, 'eleven': 11}
```

Or to change the value associated with an existing key

In [33]:
```
numbers['one'] = 'Pluto'
numbers
```

Out[33]:
```
{'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}
```

Python has *dictionary comprehensions* with a syntax similar to the list comprehensions we saw in the previous tutorial.

In [34]:
```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus
planet_to_initial = {planet: planet[0] for planet in planets}
planet_to_initial
```

Out[34]:
```
{'Mercury': 'M',
 'Venus': 'V',
 'Earth': 'E',
 'Mars': 'M',
 'Jupiter': 'J',
 'Saturn': 'S',
 'Uranus': 'U',
 'Neptune': 'N'}
```

The `in` operator tells us whether something is a key in the dictionary

In [35]:
```
'Saturn' in planet_to_initial
```

Out[35]:
```
True
```

In [36]:
```
'Betelgeuse' in planet_to_initial
```

Out[36]:
```
False
```

A for loop over a dictionary will loop over its keys

In [37]:
```
for k in numbers:
    print("{} = {}".format(k, numbers[k]))
```

```
one = Pluto
two = 2
three = 3
eleven = 11
```

We can access a collection of all the keys or all the values with `dict.keys()` and `dict.values()`, respectively.

In [38]:
```
# Get all the initials, sort them alphabetically, and put them in a space-se
' '.join(sorted(planet_to_initial.values()))
```

Loading [MathJax]/extensions/Safe.js

```
S U V'
```

The very useful `dict.items()` method lets us iterate over the keys and values of a dictionary simultaneously. (In Python jargon, an **item** refers to a key, value pair)

In [39]:
```
for planet, initial in planet_to_initial.items():
    print("{} begins with \"{}\"".format(planet.rjust(10), initial))
```

```
 Mercury begins with "M"
   Venus begins with "V"
   Earth begins with "E"
    Mars begins with "M"
 Jupiter begins with "J"
  Saturn begins with "S"
  Uranus begins with "U"
 Neptune begins with "N"
```

To read a full inventory of dictionaries' methods, click the "output" button below to read the full help page, or check out the official online documentation.

In [40]:
```
help(dict)
```

Loading [MathJax]/extensions/Safe.js

```
Help on class dict in module builtins:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Built-in subclasses:
 |      StgDict
 |
 |  Methods defined here:
 |
 |  __contains__(self, key, /)
 |      True if the dictionary has the specified key, else False.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __reversed__(self, /)
 |      Return a reverse iterator over the dict keys.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(...)
 |      D.clear() -> None.  Remove all items from D.
 |
 |  copy(...)
 |      D.copy() -> a shallow copy of D
 |
 |  get(self, key, default=None, /)
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  items(...)
 |      D.items() -> a set-like object providing a view on D's items
 |
 |  keys(...)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(...)
 |      D.pop(k[,d]) -> v, remove specified key and return the corresponding
value.
 |
 |      If the key is not found, return the default if given; otherwise,
 |      raise a KeyError.
 |
 |  popitem(self, /)
 |      Remove and return a (key, value) pair as a 2-tuple.
 |
 |      Pairs are returned in LIFO (last-in, first-out) order.
 |      Raises KeyError if the dict is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(...)
```

Loading [MathJax]/extensions/Safe.js

```
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D
[k] = E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in
E: D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  __class_getitem__(...) from builtins.type
 |      See PEP 585
 |
 |  fromkeys(iterable, value=None, /) from builtins.type
 |      Create a new dictionary with keys from iterable and values set to va
lue.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signatu
re.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Your Turn

You've learned a lot of Python... go **demonstrate your new skills** with some realistic programming applications.

Loading [MathJax]/extensions/Safe.js

In this tutorial, you will learn about **imports** in Python, get some tips for working with unfamiliar libraries (and the objects they return), and dig into **operator overloading**.

Imports

So far we've talked about types and functions which are built-in to the language.

But one of the best things about Python (especially if you're a data scientist) is the vast number of high-quality custom libraries that have been written for it.

Some of these libraries are in the "standard library", meaning you can find them anywhere you run Python. Other libraries can be easily added, even if they aren't always shipped with Python.

Either way, we'll access this code with **imports**.

We'll start our example by importing `math` from the standard library.

In [1]:
```
import math

print("It's math! It has type {}".format(type(math)))
```

```
It's math! It has type <class 'module'>
```

`math` is a module. A module is just a collection of variables (a *namespace*, if you like) defined by someone else. We can see all the names in `math` using the built-in function `dir()`.

In [2]:
```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil',
'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'e
xp', 'exp2', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'is
qrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'na
n', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

We can access these variables using dot syntax. Some of them refer to simple values, like `math.pi`:

In [3]:
```
print("pi to 4 significant digits = {:.4}".format(math.pi))
```

```
pi to 4 significant digits = 3.142
```

But most of what we'll find in the module are functions, like `math.log`:

Loading [MathJax]/extensions/Safe.js

In [4]:
```
math.log(32, 2)
```

Out[4]: 5.0

Of course, if we don't know what `math.log` does, we can call `help()` on it:

In [5]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

We can also call `help()` on the module itself. This will give us the combined documentation for *all* the functions and values in the module (as well as a high-level description of the module). Click the "output" button to see the whole `math` help page.

In [6]:
```
help(math)
```

Loading [MathJax]/extensions/Safe.js

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.11/library/math.html

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    cbrt(x, /)
        Return the cube root of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.
```

Loading [MathJax]/extensions/Safe.js

```
    comb(n, k, /)
        Number of ways to choose k items from n items without repetition and
without order.

        Evaluates to n! / (k! * (n - k)!) when k <= n and evaluates
        to zero when k > n.

        Also called the binomial coefficient because it is equivalent
        to the coefficient of k-th term in polynomial expansion of the
        expression (1 + x)**n.

        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.

    copysign(x, y, /)
        Return a float with the magnitude (absolute value) of x but the sign
of y.

        On platforms that support signed zeros, copysign(1.0, -0.0)
        returns -1.0.

    cos(x, /)
        Return the cosine of x (measured in radians).

    cosh(x, /)
        Return the hyperbolic cosine of x.

    degrees(x, /)
        Convert angle x from radians to degrees.

    dist(p, q, /)
        Return the Euclidean distance between two points p and q.

        The points should be specified as sequences (or iterables) of
        coordinates.  Both inputs must have the same dimension.

        Roughly equivalent to:
            sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))

    erf(x, /)
        Error function at x.

    erfc(x, /)
        Complementary error function at x.

    exp(x, /)
        Return e raised to the power of x.

    exp2(x, /)
        Return 2 raised to the power of x.

    expm1(x, /)
        Return exp(x)-1.

        This function avoids the loss of precision involved in the direct ev
aluation of exp(x)-1 for small x.
```

Loading [MathJax]/extensions/Safe.js

```
fabs(x, /)
    Return the absolute value of the float x.

factorial(n, /)
    Find n!.

    Raise a ValueError if x is negative or non-integral.

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.

fmod(x, y, /)
    Return fmod(x, y), according to platform C.

    x % y may differ.

frexp(x, /)
    Return the mantissa and exponent of x, as pair (m, e).

    m is a float and e is an int, such that x = m * 2.**e.
    If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

fsum(seq, /)
    Return an accurate floating point sum of values in the iterable seq.

    Assumes IEEE-754 floating point arithmetic.

gamma(x, /)
    Gamma function at x.

gcd(*integers)
    Greatest Common Divisor.

hypot(...)
    hypot(*coordinates) -> value

    Multidimensional Euclidean distance from the origin to a point.

    Roughly equivalent to:
        sqrt(sum(x**2 for x in coordinates))

    For a two dimensional point (x, y), gives the hypotenuse
    using the Pythagorean theorem:  sqrt(x*x + y*y).

    For example, the hypotenuse of a 3/4/5 right triangle is:

        >>> hypot(3.0, 4.0)
        5.0

isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
    Determine whether two floating point numbers are close in value.
```

Loading [MathJax]/extensions/Safe.js

```
            rel_tol
```

```
            maximum difference for being considered "close", relative to the
            magnitude of the input values
          abs_tol
            maximum difference for being considered "close", regardless of t
he
            magnitude of the input values

        Return True if a is close in value to b, and False otherwise.

        For the values to be considered close, the difference between them
        must be smaller than at least one of the tolerances.

        -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
        is, NaN is not close to anything, even itself.  inf and -inf are
        only close to themselves.

    isfinite(x, /)
        Return True if x is neither an infinity nor a NaN, and False otherwi
se.

    isinf(x, /)
        Return True if x is a positive or negative infinity, and False other
wise.

    isnan(x, /)
        Return True if x is a NaN (not a number), and False otherwise.

    isqrt(n, /)
        Return the integer part of the square root of the input.

    lcm(*integers)
        Least Common Multiple.

    ldexp(x, i, /)
        Return x * (2**i).

        This is essentially the inverse of frexp().

    lgamma(x, /)
        Natural logarithm of absolute value of Gamma function at x.

    log(...)
        log(x, [base=math.e])
        Return the logarithm of x to the given base.

        If the base not specified, returns the natural logarithm (base e) of
x.

    log10(x, /)
        Return the base 10 logarithm of x.

    log1p(x, /)
        Return the natural logarithm of 1+x (base e).

        The result is computed in a way which is accurate for x near zero.
```

Loading [MathJax]/extensions/Safe.js

```
    log2(x, /)
        Return the base 2 logarithm of x.

    modf(x, /)
        Return the fractional and integer parts of x.

        Both results carry the sign of x and are floats.

    nextafter(x, y, /)
        Return the next floating-point value after x towards y.

    perm(n, k=None, /)
        Number of ways to choose k items from n items without repetition and
with order.

        Evaluates to n! / (n - k)! when k <= n and evaluates
        to zero when k > n.

        If k is not specified or is None, then k defaults to n
        and the function returns n!.

        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.

    pow(x, y, /)
        Return x**y (x to the power of y).

    prod(iterable, /, *, start=1)
        Calculate the product of all the elements in the input iterable.

        The default start value for the product is 1.

        When the iterable is empty, return the start value.  This function i
s
        intended specifically for use with numeric values and may reject
        non-numeric types.

    radians(x, /)
        Convert angle x from degrees to radians.

    remainder(x, y, /)
        Difference between x and the closest integer multiple of y.

        Return x - n*y where n*y is the closest integer multiple of y.
        In the case where x is exactly halfway between two multiples of
        y, the nearest even value of n is used. The result is always exact.

    sin(x, /)
        Return the sine of x (measured in radians).

    sinh(x, /)
        Return the hyperbolic sine of x.

    sqrt(x, /)
        Return the square root of x.
```

Loading [MathJax]/extensions/Safe.js

```
    tan(x, /)
        Return the tangent of x (measured in radians).

    tanh(x, /)
        Return the hyperbolic tangent of x.

    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.

        Uses the __trunc__ magic method.

    ulp(x, /)
        Return the value of the least significant bit of the float x.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /Users/akarate/.pyenv/versions/3.11.5/Library/Frameworks/Python.framewor
k/Versions/3.11/lib/python3.11/lib-dynload/math.cpython-311-darwin.so
```

Other import syntax

If we know we'll be using functions in `math` frequently we can import it under a shorter alias to save some typing (though in this case "math" is already pretty short).

In [7]:
```
import math as mt
mt.pi
```

Out[7]:
```
3.141592653589793
```

> You may have seen code that does this with certain popular libraries like Pandas, Numpy, Tensorflow, or Matplotlib. For example, it's a common convention to `import numpy as np` and `import pandas as pd`.

The `as` simply renames the imported module. It's equivalent to doing something like:

In [8]:
```
import math
mt = math
```

Wouldn't it be great if we could refer to all the variables in the `math` module by themselves? i.e. if we could just refer to `pi` instead of `math.pi` or `mt.pi`? Good news: we can do that.

Loading [MathJax]/extensions/Safe.js

In [9]:
```
from math import *
print(pi, log(32, 2))
```

```
3.141592653589793 5.0
```

`import *` makes all the module's variables directly accessible to you (without any dotted prefix).

Bad news: some purists might grumble at you for doing this.

Worse: they kind of have a point.

In [10]:
```
from math import *
from numpy import *
print(pi, log(32, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 3
      1 from math import *
      2 from numpy import *
----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType
```

What has happened? It worked before!

These kinds of "star imports" can occasionally lead to weird, difficult-to-debug situations.

The problem in this case is that the `math` and `numpy` modules both have functions called `log`, but they have different semantics. Because we import from `numpy` second, its `log` overwrites (or "shadows") the `log` variable we imported from `math`.

A good compromise is to import only the specific things we'll need from each module:

In [11]:
```
from math import log, pi
from numpy import asarray
```

Submodules

We've seen that modules contain variables which can refer to functions or values. Something to be aware of is that they can also have variables referring to *other modules*.

In [12]:
```
import numpy
print("numpy.random is a", type(numpy.random))
print("it contains names such as...",
      dir(numpy.random)[-15:]
     )
```

Loading [MathJax]/extensions/Safe.js

```
numpy.random is a <class 'module'>
it contains names such as... ['set_bit_generator', 'set_state', 'shuffle',
'standard_cauchy', 'standard_exponential', 'standard_gamma', 'standard_norma
l', 'standard_t', 'test', 'triangular', 'uniform', 'vonmises', 'wald', 'weib
ull', 'zipf']
```

So if we import `numpy` as above, then calling a function in the `random` "submodule" will require *two* dots.

In [13]:
```
# Roll 10 dice
rolls = numpy.random.randint(low=1, high=6, size=10)
rolls
```

Out[13]:
```
array([1, 1, 4, 3, 2, 1, 5, 1, 2, 2])
```

Oh the places you'll go, oh the objects you'll see

So after 6 lessons, you're a pro with ints, floats, bools, lists, strings, and dicts (right?).

Even if that were true, it doesn't end there. As you work with various libraries for specialized tasks, you'll find that they define their own types which you'll have to learn to work with. For example, if you work with the graphing library `matplotlib`, you'll be coming into contact with objects it defines which represent Subplots, Figures, TickMarks, and Annotations. `pandas` functions will give you DataFrames and Series.

In this section, I want to share with you a quick survival guide for working with strange types.

Three tools for understanding strange objects

In the cell above, we saw that calling a `numpy` function gave us an "array". We've never seen anything like this before (not in this course anyways). But don't panic: we have three familiar builtin functions to help us here.

1: `type()` (what is this thing?)

In [14]:
```
type(rolls)
```

Out[14]:
```
numpy.ndarray
```

2: `dir()` (what can I do with it?)

In [15]:
```
print(dir(rolls))
```

Loading [MathJax]/extensions/Safe.js

```
['T', '__abs__', '__add__', '__and__', '__array__', '__array_finalize__', '_
_array_function__', '__array_interface__', '__array_prepare__', '__array_pri
ority__', '__array_struct__', '__array_ufunc__', '__array_wrap__', '__bool_
_', '__class__', '__class_getitem__', '__complex__', '__contains__', '__copy
__', '__deepcopy__', '__delattr__', '__delitem__', '__dir__', '__divmod__',
'__dlpack__', '__dlpack_device__', '__doc__', '__eq__', '__float__', '__floo
rdiv__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getst
ate__', '__gt__', '__hash__', '__iadd__', '__iand__', '__ifloordiv__', '__il
shift__', '__imatmul__', '__imod__', '__imul__', '__index__', '__init__', '_
_init_subclass__', '__int__', '__invert__', '__ior__', '__ipow__', '__irshif
t__', '__isub__', '__iter__', '__itruediv__', '__ixor__', '__le__', '__len_
_', '__lshift__', '__lt__', '__matmul__', '__mod__', '__mul__', '__ne__', '_
_neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmatmul__', '__rmod__', '__rmul__', '__ror__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__seta
ttr__', '__setitem__', '__setstate__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__xor__', 'all', 'any', 'argmax', 'argmi
n', 'argpartition', 'argsort', 'astype', 'base', 'byteswap', 'choose', 'cli
p', 'compress', 'conj', 'conjugate', 'copy', 'ctypes', 'cumprod', 'cumsum',
'data', 'diagonal', 'dot', 'dtype', 'dump', 'dumps', 'fill', 'flags', 'fla
t', 'flatten', 'getfield', 'imag', 'item', 'itemset', 'itemsize', 'max', 'me
an', 'min', 'nbytes', 'ndim', 'newbyteorder', 'nonzero', 'partition', 'pro
d', 'ptp', 'put', 'ravel', 'real', 'repeat', 'reshape', 'resize', 'round',
'searchsorted', 'setfield', 'setflags', 'shape', 'size', 'sort', 'squeeze',
'std', 'strides', 'sum', 'swapaxes', 'take', 'tobytes', 'tofile', 'tolist',
'tostring', 'trace', 'transpose', 'var', 'view']
```

In [16]:
```
# If I want the average roll, the "mean" method looks promising...
rolls.mean()
```

Out[16]:
```
2.2
```

In [17]:
```
# Or maybe I just want to turn the array into a list, in which case I can us
rolls.tolist()
```

Out[17]:
```
[1, 1, 4, 3, 2, 1, 5, 1, 2, 2]
```

3: `help()` (tell me more)

In [18]:
```
# That "ravel" attribute sounds interesting. I'm a big classical music fan.
help(rolls.ravel)
```

Loading [MathJax]/extensions/Safe.js

```
Help on built-in function ravel:

ravel(...) method of numpy.ndarray instance
    a.ravel([order])

    Return a flattened array.

    Refer to `numpy.ravel` for full documentation.

    See Also
    --------
    numpy.ravel : equivalent function

    ndarray.flat : a flat iterator on the array.
```

In [19]:
```
# Okay, just tell me everything there is to know about numpy.ndarray
# (Click the "output" button to see the novel-length output)
help(rolls)
```

Loading [MathJax]/extensions/Safe.js

```
Help on ndarray object:

class ndarray(builtins.object)
 |  ndarray(shape, dtype=float, buffer=None, offset=0,
 |          strides=None, order=None)
 |
 |  An array object represents a multidimensional, homogeneous array
 |  of fixed-size items.  An associated data-type object describes the
 |  format of each element in the array (its byte-order, how many bytes it
 |  occupies in memory, whether it is an integer, a floating point number,
 |  or something else, etc.)
 |
 |  Arrays should be constructed using `array`, `zeros` or `empty` (refer
 |  to the See Also section below).  The parameters given here refer to
 |  a low-level method (`ndarray(...)`) for instantiating an array.
 |
 |  For more information, refer to the `numpy` module and examine the
 |  methods and attributes of an array.
 |
 |  Parameters
 |  ----------
 |  (for the __new__ method; see Notes below)
 |
 |  shape : tuple of ints
 |      Shape of created array.
 |  dtype : data-type, optional
 |      Any object that can be interpreted as a numpy data type.
 |  buffer : object exposing buffer interface, optional
 |      Used to fill the array with data.
 |  offset : int, optional
 |      Offset of array data in buffer.
 |  strides : tuple of ints, optional
 |      Strides of data in memory.
 |  order : {'C', 'F'}, optional
 |      Row-major (C-style) or column-major (Fortran-style) order.
 |
 |  Attributes
 |  ----------
 |  T : ndarray
 |      Transpose of the array.
 |  data : buffer
 |      The array's elements, in memory.
 |  dtype : dtype object
 |      Describes the format of the elements in the array.
 |  flags : dict
 |      Dictionary containing information related to memory use, e.g.,
 |      'C_CONTIGUOUS', 'OWNDATA', 'WRITEABLE', etc.
 |  flat : numpy.flatiter object
 |      Flattened version of the array as an iterator.  The iterator
 |      allows assignments, e.g., ``x.flat = 3`` (See `ndarray.flat` for
 |      assignment examples; TODO).
 |  imag : ndarray
 |      Imaginary part of the array.
 |  real : ndarray
 |      Real part of the array.
 |  size : int
```

Loading [MathJax]/extensions/Safe.js

```
|      Number of elements in the array.
|  itemsize : int
|      The memory use of each array element in bytes.
|  nbytes : int
|      The total number of bytes required to store the array data,
|      i.e., ``itemsize * size``.
|  ndim : int
|      The array's number of dimensions.
|  shape : tuple of ints
|      Shape of the array.
|  strides : tuple of ints
|      The step-size required to move from one element to the next in
|      memory. For example, a contiguous ``(3, 4)`` array of type
|      ``int16`` in C-order has strides ``(8, 2)``.  This implies that
|      to move from element to element in memory requires jumps of 2 bytes.
|      To move from row-to-row, one needs to jump 8 bytes at a time
|      (``2 * 4``).
|  ctypes : ctypes object
|      Class containing properties of the array needed for interaction
|      with ctypes.
|  base : ndarray
|      If the array is a view into another array, that array is its `base`
|      (unless that array is also a view).  The `base` array is where the
|      array data is actually stored.
|
|  See Also
|  --------
|  array : Construct an array.
|  zeros : Create an array, each element of which is zero.
|  empty : Create an array, but leave its allocated memory unchanged (i.e.,
|          it contains "garbage").
|  dtype : Create a data-type.
|  numpy.typing.NDArray : An ndarray alias :term:`generic <generic type>`
|                         w.r.t. its `dtype.type <numpy.dtype.type>`.
|
|  Notes
|  -----
|  There are two modes of creating an array using ``__new__``:
|
|  1. If `buffer` is None, then only `shape`, `dtype`, and `order`
|     are used.
|  2. If `buffer` is an object exposing the buffer interface, then
|     all keywords are interpreted.
|
|  No ``__init__`` method is needed because the array is fully initialized
|  after the ``__new__`` method.
|
|  Examples
|  --------
|  These examples illustrate the low-level `ndarray` constructor.  Refer
|  to the `See Also` section above for easier ways of constructing an
|  ndarray.
|
|  First mode, `buffer` is None:
|
|  >>> np.ndarray(shape=(2,2), dtype=float, order='F')
```

Loading [MathJax]/extensions/Safe.js

```
 |  array([[0.0e+000, 0.0e+000], # random
 |         [     nan, 2.5e-323]])
 |
 |  Second mode:
 |
 |  >>> np.ndarray((2,), buffer=np.array([1,2,3]),
 |  ...            offset=np.int_().itemsize,
 |  ...            dtype=int) # offset = 1*itemsize, i.e. skip first element
 |  array([2, 3])
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
 |  __array__(...)
 |      a.__array__([dtype], /)
 |
 |      Returns either a new reference to self if dtype is not given or a ne
w array
 |      of provided data type if dtype is different from the current dtype o
f the
 |      array.
 |
 |  __array_finalize__(...)
 |      a.__array_finalize__(obj, /)
 |
 |      Present so subclasses can call super. Does nothing.
 |
 |  __array_function__(...)
 |
 |  __array_prepare__(...)
 |      a.__array_prepare__(array[, context], /)
 |
 |      Returns a view of `array` with the same type as self.
 |
 |  __array_ufunc__(...)
 |
 |  __array_wrap__(...)
 |      a.__array_wrap__(array[, context], /)
 |
 |      Returns a view of `array` with the same type as self.
 |
 |  __bool__(self, /)
 |      True if self else False
 |
 |  __complex__(...)
 |
 |  __contains__(self, key, /)
 |      Return key in self.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |  __copy__(...)
 |      a.__copy__()
 |
 |      Used if :func:`copy.copy` is called on an array. Returns a copy of t
he array.
 |
 |      Equivalent to ``a.copy(order='K')``.
 |
 |  __deepcopy__(...)
 |      a.__deepcopy__(memo, /)
 |
 |      Used if :func:`copy.deepcopy` is called on an array.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __dlpack__(...)
 |      a.__dlpack__(*, stream=None)
 |
 |      DLPack Protocol: Part of the Array API.
 |
 |  __dlpack_device__(...)
 |      a.__dlpack_device__()
 |
 |      DLPack Protocol: Part of the Array API.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(...)
 |      Default object formatter.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Return self+=value.
 |
 |  __iand__(self, value, /)
 |      Return self&=value.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |  __ifloordiv__(self, value, /)
 |      Return self//=value.
 |
 |  __ilshift__(self, value, /)
 |      Return self<<=value.
 |
 |  __imatmul__(self, value, /)
 |      Return self@=value.
 |
 |  __imod__(self, value, /)
 |      Return self%=value.
 |
 |  __imul__(self, value, /)
 |      Return self*=value.
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as
an index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __ior__(self, value, /)
 |      Return self|=value.
 |
 |  __ipow__(self, value, /)
 |      Return self**=value.
 |
 |  __irshift__(self, value, /)
 |      Return self>>=value.
 |
 |  __isub__(self, value, /)
 |      Return self-=value.
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __itruediv__(self, value, /)
 |      Return self/=value.
 |
 |  __ixor__(self, value, /)
 |      Return self^=value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
```

Loading [MathJax]/extensions/Safe.js

```
|  __lt__(self, value, /)
|      Return self<value.
|
|  __matmul__(self, value, /)
|      Return self@value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __reduce__(...)
|      a.__reduce__()
|
|      For pickling.
|
|  __reduce_ex__(...)
|      Helper for pickle.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmatmul__(self, value, /)
|      Return value@self.
|
```

Loading [MathJax]/extensions/Safe.js

```
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __rpow__(self, value, mod=None, /)
 |      Return pow(value, self, mod).
 |
 |  __rrshift__(self, value, /)
 |      Return value>>self.
 |
 |  __rshift__(self, value, /)
 |      Return self>>value.
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __setstate__(...)
 |      a.__setstate__(state, /)
 |
 |      For unpickling.
 |
 |      The `state` argument must be a sequence that contains the following
 |      elements:
 |
 |      Parameters
 |      ----------
 |      version : int
 |          optional pickle version. If omitted defaults to 0.
 |      shape : tuple
 |      dtype : data-type
 |      isFortran : bool
 |      rawdata : string or list
 |          a binary string with the data (or a list if 'a' is an object arr
ay)
 |
 |  __sizeof__(...)
 |      Size of object in memory, in bytes.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  __sub__(self, value, /)
```

Loading [MathJax]/extensions/Safe.js

```
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  all(...)
|      a.all(axis=None, out=None, keepdims=False, *, where=True)
|
|      Returns True if all elements evaluate to True.
|
|      Refer to `numpy.all` for full documentation.
|
|      See Also
|      --------
|      numpy.all : equivalent function
|
|  any(...)
|      a.any(axis=None, out=None, keepdims=False, *, where=True)
|
|      Returns True if any of the elements of `a` evaluate to True.
|
|      Refer to `numpy.any` for full documentation.
|
|      See Also
|      --------
|      numpy.any : equivalent function
|
|  argmax(...)
|      a.argmax(axis=None, out=None, *, keepdims=False)
|
|      Return indices of the maximum values along the given axis.
|
|      Refer to `numpy.argmax` for full documentation.
|
|      See Also
|      --------
|      numpy.argmax : equivalent function
|
|  argmin(...)
|      a.argmin(axis=None, out=None, *, keepdims=False)
|
|      Return indices of the minimum values along the given axis.
|
|      Refer to `numpy.argmin` for detailed documentation.
|
|      See Also
|      --------
|      numpy.argmin : equivalent function
|
|  argpartition(...)
|      a.argpartition(kth, axis=-1, kind='introselect', order=None)
|
|      Returns the indices that would partition this array.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |      Refer to `numpy.argpartition` for full documentation.
 |
 |      .. versionadded:: 1.8.0
 |
 |      See Also
 |      --------
 |      numpy.argpartition : equivalent function
 |
 |  argsort(...)
 |      a.argsort(axis=-1, kind=None, order=None)
 |
 |      Returns the indices that would sort this array.
 |
 |      Refer to `numpy.argsort` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.argsort : equivalent function
 |
 |  astype(...)
 |      a.astype(dtype, order='K', casting='unsafe', subok=True, copy=True)
 |
 |      Copy of the array, cast to a specified type.
 |
 |      Parameters
 |      ----------
 |      dtype : str or dtype
 |          Typecode or data-type to which the array is cast.
 |      order : {'C', 'F', 'A', 'K'}, optional
 |          Controls the memory layout order of the result.
 |          'C' means C order, 'F' means Fortran order, 'A'
 |          means 'F' order if all the arrays are Fortran contiguous,
 |          'C' order otherwise, and 'K' means as close to the
 |          order the array elements appear in memory as possible.
 |          Default is 'K'.
 |      casting : {'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional
 |          Controls what kind of data casting may occur. Defaults to 'unsaf
e'
 |          for backwards compatibility.
 |
 |            * 'no' means the data types should not be cast at all.
 |            * 'equiv' means only byte-order changes are allowed.
 |            * 'safe' means only casts which can preserve values are allowe
d.
 |            * 'same_kind' means only safe casts or casts within a kind,
 |              like float64 to float32, are allowed.
 |            * 'unsafe' means any data conversions may be done.
 |      subok : bool, optional
 |          If True, then sub-classes will be passed-through (default), othe
rwise
 |          the returned array will be forced to be a base-class array.
 |      copy : bool, optional
 |          By default, astype always returns a newly allocated array. If th
is
 |          is set to false, and the `dtype`, `order`, and `subok`
```

Loading [MathJax]/extensions/Safe.js

```
 |          requirements are satisfied, the input array is returned instead
 |          of a copy.
 |
 |      Returns
 |      -------
 |      arr_t : ndarray
 |          Unless `copy` is False and the other conditions for returning th
e input
 |          array are satisfied (see description for `copy` input paramete
r), `arr_t`
 |          is a new array of the same shape as the input array, with dtype,
order
 |          given by `dtype`, `order`.
 |
 |      Notes
 |      -----
 |      .. versionchanged:: 1.17.0
 |         Casting between a simple data type and a structured one is possib
le only
 |         for "unsafe" casting.  Casting to multiple fields is allowed, but
 |         casting from multiple fields is not.
 |
 |      .. versionchanged:: 1.9.0
 |         Casting from numeric to string types in 'safe' casting mode requi
res
 |         that the string dtype length is long enough to store the max
 |         integer/float value converted.
 |
 |      Raises
 |      ------
 |      ComplexWarning
 |          When casting from complex to float or int. To avoid this,
 |          one should use ``a.real.astype(t)``.
 |
 |      Examples
 |      --------
 |      >>> x = np.array([1, 2, 2.5])
 |      >>> x
 |      array([1. ,  2. ,  2.5])
 |
 |      >>> x.astype(int)
 |      array([1, 2, 2])
 |
 |  byteswap(...)
 |      a.byteswap(inplace=False)
 |
 |      Swap the bytes of the array elements
 |
 |      Toggle between low-endian and big-endian data representation by
 |      returning a byteswapped array, optionally swapped in-place.
 |      Arrays of byte-strings are not swapped. The real and imaginary
 |      parts of a complex number are swapped individually.
 |
 |      Parameters
 |      ----------
 |      inplace : bool, optional
```

Loading [MathJax]/extensions/Safe.js

```
 |          If ``True``, swap bytes in-place, default is ``False``.
 |
 |      Returns
 |      -------
 |      out : ndarray
 |          The byteswapped array. If `inplace` is ``True``, this is
 |          a view to self.
 |
 |      Examples
 |      --------
 |      >>> A = np.array([1, 256, 8755], dtype=np.int16)
 |      >>> list(map(hex, A))
 |      ['0x1', '0x100', '0x2233']
 |      >>> A.byteswap(inplace=True)
 |      array([  256,     1, 13090], dtype=int16)
 |      >>> list(map(hex, A))
 |      ['0x100', '0x1', '0x3322']
 |
 |      Arrays of byte-strings are not swapped
 |
 |      >>> A = np.array([b'ceg', b'fac'])
 |      >>> A.byteswap()
 |      array([b'ceg', b'fac'], dtype='|S3')
 |
 |      ``A.newbyteorder().byteswap()`` produces an array with the same valu
es
 |        but different representation in memory
 |
 |      >>> A = np.array([1, 2, 3])
 |      >>> A.view(np.uint8)
 |      array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0,
0, 0,
 |             0, 0], dtype=uint8)
 |      >>> A.newbyteorder().byteswap(inplace=True)
 |      array([1, 2, 3])
 |      >>> A.view(np.uint8)
 |      array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,
0, 0,
 |             0, 3], dtype=uint8)
 |
 |  choose(...)
 |      a.choose(choices, out=None, mode='raise')
 |
 |      Use an index array to construct a new array from a set of choices.
 |
 |      Refer to `numpy.choose` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.choose : equivalent function
 |
 |  clip(...)
 |      a.clip(min=None, max=None, out=None, **kwargs)
 |
 |      Return an array whose values are limited to ``[min, max]``.
 |      One of max or min must be given.
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |      Refer to `numpy.clip` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.clip : equivalent function
 |
 |  compress(...)
 |      a.compress(condition, axis=None, out=None)
 |
 |      Return selected slices of this array along given axis.
 |
 |      Refer to `numpy.compress` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.compress : equivalent function
 |
 |  conj(...)
 |      a.conj()
 |
 |      Complex-conjugate all elements.
 |
 |      Refer to `numpy.conjugate` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.conjugate : equivalent function
 |
 |  conjugate(...)
 |      a.conjugate()
 |
 |      Return the complex conjugate, element-wise.
 |
 |      Refer to `numpy.conjugate` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.conjugate : equivalent function
 |
 |  copy(...)
 |      a.copy(order='C')
 |
 |      Return a copy of the array.
 |
 |      Parameters
 |      ----------
 |      order : {'C', 'F', 'A', 'K'}, optional
 |          Controls the memory layout of the copy. 'C' means C-order,
 |          'F' means F-order, 'A' means 'F' if `a` is Fortran contiguous,
 |          'C' otherwise. 'K' means match the layout of `a` as closely
 |          as possible. (Note that this function and :func:`numpy.copy` are
very
 |          similar but have different default values for their order=
 |          arguments, and this function always passes sub-classes through.)
 |
```

Loading [MathJax]/extensions/Safe.js

```
 |      See also
 |      --------
 |      numpy.copy : Similar function with different default behavior
 |      numpy.copyto
 |
 |      Notes
 |      -----
 |      This function is the preferred method for creating an array copy.  T
he
 |      function :func:`numpy.copy` is similar, but it defaults to using ord
er 'K',
 |      and will not pass sub-classes through by default.
 |
 |      Examples
 |      --------
 |      >>> x = np.array([[1,2,3],[4,5,6]], order='F')
 |
 |      >>> y = x.copy()
 |
 |      >>> x.fill(0)
 |
 |      >>> x
 |      array([[0, 0, 0],
 |             [0, 0, 0]])
 |
 |      >>> y
 |      array([[1, 2, 3],
 |             [4, 5, 6]])
 |
 |      >>> y.flags['C_CONTIGUOUS']
 |      True
 |
 |  cumprod(...)
 |      a.cumprod(axis=None, dtype=None, out=None)
 |
 |      Return the cumulative product of the elements along the given axis.
 |
 |      Refer to `numpy.cumprod` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.cumprod : equivalent function
 |
 |  cumsum(...)
 |      a.cumsum(axis=None, dtype=None, out=None)
 |
 |      Return the cumulative sum of the elements along the given axis.
 |
 |      Refer to `numpy.cumsum` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.cumsum : equivalent function
 |
 |  diagonal(...)
 |      a.diagonal(offset=0, axis1=0, axis2=1)
```

Loading [MathJax]/extensions/Safe.js

```
|
|      Return specified diagonals. In NumPy 1.9 the returned array is a
|      read-only view instead of a copy as in previous NumPy versions.  In
|      a future version the read-only restriction will be removed.
|
|      Refer to :func:`numpy.diagonal` for full documentation.
|
|      See Also
|      --------
|      numpy.diagonal : equivalent function
|
|  dot(...)
|
|  dump(...)
|      a.dump(file)
|
|      Dump a pickle of the array to the specified file.
|      The array can be read back with pickle.load or numpy.load.
|
|      Parameters
|      ----------
|      file : str or Path
|          A string naming the dump file.
|
|          .. versionchanged:: 1.17.0
|              `pathlib.Path` objects are now accepted.
|
|  dumps(...)
|      a.dumps()
|
|      Returns the pickle of the array as a string.
|      pickle.loads will convert the string back to an array.
|
|      Parameters
|      ----------
|      None
|
|  fill(...)
|      a.fill(value)
|
|      Fill the array with a scalar value.
|
|      Parameters
|      ----------
|      value : scalar
|          All elements of `a` will be assigned this value.
|
|      Examples
|      --------
|      >>> a = np.array([1, 2])
|      >>> a.fill(0)
|      >>> a
|      array([0, 0])
|      >>> a = np.empty(2)
|      >>> a.fill(1)
|      >>> a
```

Loading [MathJax]/extensions/Safe.js

```
 |      array([1.,  1.])
 |
 |      Fill expects a scalar value and always behaves the same as assigning
 |      to a single array element.  The following is a rare example where th
is
 |      distinction is important:
 |
 |      >>> a = np.array([None, None], dtype=object)
 |      >>> a[0] = np.array(3)
 |      >>> a
 |      array([array(3), None], dtype=object)
 |      >>> a.fill(np.array(3))
 |      >>> a
 |      array([array(3), array(3)], dtype=object)
 |
 |      Where other forms of assignments will unpack the array being assigne
d:
 |
 |      >>> a[...] = np.array(3)
 |      >>> a
 |      array([3, 3], dtype=object)
 |
 |  flatten(...)
 |      a.flatten(order='C')
 |
 |      Return a copy of the array collapsed into one dimension.
 |
 |      Parameters
 |      ----------
 |      order : {'C', 'F', 'A', 'K'}, optional
 |          'C' means to flatten in row-major (C-style) order.
 |          'F' means to flatten in column-major (Fortran-
 |          style) order. 'A' means to flatten in column-major
 |          order if `a` is Fortran *contiguous* in memory,
 |          row-major order otherwise. 'K' means to flatten
 |          `a` in the order the elements occur in memory.
 |          The default is 'C'.
 |
 |      Returns
 |      -------
 |      y : ndarray
 |          A copy of the input array, flattened to one dimension.
 |
 |      See Also
 |      --------
 |      ravel : Return a flattened array.
 |      flat : A 1-D flat iterator over the array.
 |
 |      Examples
 |      --------
 |      >>> a = np.array([[1,2], [3,4]])
 |      >>> a.flatten()
 |      array([1, 2, 3, 4])
 |      >>> a.flatten('F')
 |      array([1, 3, 2, 4])
 |
```

Loading [MathJax]/extensions/Safe.js

```
 |  getfield(...)
 |      a.getfield(dtype, offset=0)
 |
 |      Returns a field of the given array as a certain type.
 |
 |      A field is a view of the array data with a given data-type. The valu
es in
 |      the view are determined by the given type and the offset into the cu
rrent
 |      array in bytes. The offset needs to be such that the view dtype fits
in the
 |      array dtype; for example an array of dtype complex128 has 16-byte el
ements.
 |      If taking a view with a 32-bit integer (4 bytes), the offset needs t
o be
 |      between 0 and 12 bytes.
 |
 |      Parameters
 |      ----------
 |      dtype : str or dtype
 |          The data type of the view. The dtype size of the view can not be
larger
 |          than that of the array itself.
 |      offset : int
 |          Number of bytes to skip before beginning the element view.
 |
 |      Examples
 |      --------
 |      >>> x = np.diag([1.+1.j]*2)
 |      >>> x[1, 1] = 2 + 4.j
 |      >>> x
 |      array([[1.+1.j,  0.+0.j],
 |             [0.+0.j,  2.+4.j]])
 |      >>> x.getfield(np.float64)
 |      array([[1.,  0.],
 |             [0.,  2.]])
 |
 |      By choosing an offset of 8 bytes we can select the complex part of t
he
 |      array for our view:
 |
 |      >>> x.getfield(np.float64, offset=8)
 |      array([[1.,  0.],
 |             [0.,  4.]])
 |
 |  item(...)
 |      a.item(*args)
 |
 |      Copy an element of an array to a standard Python scalar and return i
t.
 |
 |      Parameters
 |      ----------
 |      \*args : Arguments (variable number and type)
 |
 |          * none: in this case, the method only works for arrays
```

Loading [MathJax]/extensions/Safe.js

```
 |            with one element (`a.size == 1`), which element is
 |            copied into a standard Python scalar object and returned.
 |
 |          * int_type: this argument is interpreted as a flat index into
 |            the array, specifying which element to copy and return.
 |
 |          * tuple of int_types: functions as does a single int_type argume
nt,
 |            except that the argument is interpreted as an nd-index into th
e
 |            array.
 |
 |      Returns
 |      -------
 |      z : Standard Python scalar object
 |          A copy of the specified element of the array as a suitable
 |          Python scalar
 |
 |      Notes
 |      -----
 |      When the data type of `a` is longdouble or clongdouble, item() retur
ns
 |      a scalar array object because there is no available Python scalar th
at
 |      would not lose information. Void arrays return a buffer object for i
tem(),
 |      unless fields are defined, in which case a tuple is returned.
 |
 |      `item` is very similar to a[args], except, instead of an array scala
r,
 |      a standard Python scalar is returned. This can be useful for speedin
g up
 |      access to elements of the array and doing arithmetic on elements of
the
 |      array using Python's optimized math.
 |
 |      Examples
 |      --------
 |      >>> np.random.seed(123)
 |      >>> x = np.random.randint(9, size=(3, 3))
 |      >>> x
 |      array([[2, 2, 6],
 |             [1, 3, 6],
 |             [1, 0, 1]])
 |      >>> x.item(3)
 |      1
 |      >>> x.item(7)
 |      0
 |      >>> x.item((0, 1))
 |      2
 |      >>> x.item((2, 2))
 |      1
 |
 |  itemset(...)
 |      a.itemset(*args)
 |
```

Loading [MathJax]/extensions/Safe.js

```
 |      Insert scalar into an array (scalar is cast to array's dtype, if pos
sible)
 |
 |      There must be at least 1 argument, and define the last argument
 |      as *item*.  Then, ``a.itemset(*args)`` is equivalent to but faster
 |      than ``a[args] = item``.  The item should be a scalar value and `arg
s`
 |      must select a single item in the array `a`.
 |
 |      Parameters
 |      ----------
 |      \*args : Arguments
 |          If one argument: a scalar, only used in case `a` is of size 1.
 |          If two arguments: the last argument is the value to be set
 |          and must be a scalar, the first argument specifies a single arra
y
 |          element location. It is either an int or a tuple.
 |
 |      Notes
 |      -----
 |      Compared to indexing syntax, `itemset` provides some speed increase
 |      for placing a scalar into a particular location in an `ndarray`,
 |      if you must do this.  However, generally this is discouraged:
 |      among other problems, it complicates the appearance of the code.
 |      Also, when using `itemset` (and `item`) inside a loop, be sure
 |      to assign the methods to a local variable to avoid the attribute
 |      look-up at each loop iteration.
 |
 |      Examples
 |      --------
 |      >>> np.random.seed(123)
 |      >>> x = np.random.randint(9, size=(3, 3))
 |      >>> x
 |      array([[2, 2, 6],
 |             [1, 3, 6],
 |             [1, 0, 1]])
 |      >>> x.itemset(4, 0)
 |      >>> x.itemset((2, 2), 9)
 |      >>> x
 |      array([[2, 2, 6],
 |             [1, 0, 6],
 |             [1, 0, 9]])
 |
 |  max(...)
 |      a.max(axis=None, out=None, keepdims=False, initial=<no value>, where
=True)
 |
 |      Return the maximum along a given axis.
 |
 |      Refer to `numpy.amax` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.amax : equivalent function
 |
 |  mean(...)
```

Loading [MathJax]/extensions/Safe.js

```
 |      a.mean(axis=None, dtype=None, out=None, keepdims=False, *, where=Tru
e)
 |
 |      Returns the average of the array elements along given axis.
 |
 |      Refer to `numpy.mean` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.mean : equivalent function
 |
 |  min(...)
 |      a.min(axis=None, out=None, keepdims=False, initial=<no value>, where
=True)
 |
 |      Return the minimum along a given axis.
 |
 |      Refer to `numpy.amin` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.amin : equivalent function
 |
 |  newbyteorder(...)
 |      arr.newbyteorder(new_order='S', /)
 |
 |      Return the array with the same data viewed with a different byte ord
er.
 |
 |      Equivalent to::
 |
 |          arr.view(arr.dtype.newbytorder(new_order))
 |
 |      Changes are also made in all fields and sub-arrays of the array data
 |      type.
 |
 |
 |
 |      Parameters
 |      ----------
 |      new_order : string, optional
 |          Byte order to force; a value from the byte order specifications
 |          below. `new_order` codes can be any of:
 |
 |          * 'S' - swap dtype from current to opposite endian
 |          * {'<', 'little'} - little endian
 |          * {'>', 'big'} - big endian
 |          * {'=', 'native'} - native order, equivalent to `sys.byteorder`
 |          * {'|', 'I'} - ignore (no change to byte order)
 |
 |          The default value ('S') results in swapping the current
 |          byte order.
 |
 |
 |      Returns
 |      -------
```

Loading [MathJax]/extensions/Safe.js

```
 |      new_arr : array
 |          New array object with the dtype reflecting given change to the
 |          byte order.
 |
 |  nonzero(...)
 |      a.nonzero()
 |
 |      Return the indices of the elements that are non-zero.
 |
 |      Refer to `numpy.nonzero` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.nonzero : equivalent function
 |
 |  partition(...)
 |      a.partition(kth, axis=-1, kind='introselect', order=None)
 |
 |      Rearranges the elements in the array in such a way that the value of
the
 |      element in kth position is in the position it would be in a sorted a
rray.
 |      All elements smaller than the kth element are moved before this elem
ent and
 |      all equal or greater are moved behind it. The ordering of the elemen
ts in
 |      the two partitions is undefined.
 |
 |      .. versionadded:: 1.8.0
 |
 |      Parameters
 |      ----------
 |      kth : int or sequence of ints
 |          Element index to partition by. The kth element value will be in
its
 |          final sorted position and all smaller elements will be moved bef
ore it
 |          and all equal or greater elements behind it.
 |          The order of all elements in the partitions is undefined.
 |          If provided with a sequence of kth it will partition all element
s
 |          indexed by kth of them into their sorted position at once.
 |
 |          .. deprecated:: 1.22.0
 |              Passing booleans as index is deprecated.
 |      axis : int, optional
 |          Axis along which to sort. Default is -1, which means sort along
the
 |          last axis.
 |      kind : {'introselect'}, optional
 |          Selection algorithm. Default is 'introselect'.
 |      order : str or list of str, optional
 |          When `a` is an array with fields defined, this argument specifie
s
 |          which fields to compare first, second, etc. A single field can
 |          be specified as a string, and not all fields need to be specifie
```

Loading [MathJax]/extensions/Safe.js

```
d,
 |          but unspecified fields will still be used, in the order in which
 |          they come up in the dtype, to break ties.
 |
 |      See Also
 |      --------
 |      numpy.partition : Return a partitioned copy of an array.
 |      argpartition : Indirect partition.
 |      sort : Full sort.
 |
 |      Notes
 |      -----
 |      See ``np.partition`` for notes on the different algorithms.
 |
 |      Examples
 |      --------
 |      >>> a = np.array([3, 4, 2, 1])
 |      >>> a.partition(3)
 |      >>> a
 |      array([2, 1, 3, 4])
 |
 |      >>> a.partition((1, 3))
 |      >>> a
 |      array([1, 2, 3, 4])
 |
 |  prod(...)
 |      a.prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, w
here=True)
 |
 |      Return the product of the array elements over the given axis
 |
 |      Refer to `numpy.prod` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.prod : equivalent function
 |
 |  ptp(...)
 |      a.ptp(axis=None, out=None, keepdims=False)
 |
 |      Peak to peak (maximum - minimum) value along a given axis.
 |
 |      Refer to `numpy.ptp` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.ptp : equivalent function
 |
 |  put(...)
 |      a.put(indices, values, mode='raise')
 |
 |      Set ``a.flat[n] = values[n]`` for all `n` in indices.
 |
 |      Refer to `numpy.put` for full documentation.
 |
 |      See Also
```

Loading [MathJax]/extensions/Safe.js

```
 |      --------
 |      numpy.put : equivalent function
 |
 |  ravel(...)
 |      a.ravel([order])
 |
 |      Return a flattened array.
 |
 |      Refer to `numpy.ravel` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.ravel : equivalent function
 |
 |      ndarray.flat : a flat iterator on the array.
 |
 |  repeat(...)
 |      a.repeat(repeats, axis=None)
 |
 |      Repeat elements of an array.
 |
 |      Refer to `numpy.repeat` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.repeat : equivalent function
 |
 |  reshape(...)
 |      a.reshape(shape, order='C')
 |
 |      Returns an array containing the same data with a new shape.
 |
 |      Refer to `numpy.reshape` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.reshape : equivalent function
 |
 |      Notes
 |      -----
 |      Unlike the free function `numpy.reshape`, this method on `ndarray` a
llows
 |      the elements of the shape parameter to be passed in as separate argu
ments.
 |      For example, ``a.reshape(10, 11)`` is equivalent to
 |      ``a.reshape((10, 11))``.
 |
 |  resize(...)
 |      a.resize(new_shape, refcheck=True)
 |
 |      Change shape and size of array in-place.
 |
 |      Parameters
 |      ----------
 |      new_shape : tuple of ints, or `n` ints
 |          Shape of resized array.
```

Loading [MathJax]/extensions/Safe.js

```
 |      refcheck : bool, optional
 |          If False, reference count will not be checked. Default is True.
 |
 |      Returns
 |      -------
 |      None
 |
 |      Raises
 |      ------
 |      ValueError
 |          If `a` does not own its own data or references or views to it ex
ist,
 |          and the data memory must be changed.
 |          PyPy only: will always raise if the data memory must be changed,
since
 |          there is no reliable way to determine if references or views to
it
 |          exist.
 |
 |      SystemError
 |          If the `order` keyword argument is specified. This behaviour is
a
 |          bug in NumPy.
 |
 |      See Also
 |      --------
 |      resize : Return a new array with the specified shape.
 |
 |      Notes
 |      -----
 |      This reallocates space for the data area if necessary.
 |
 |      Only contiguous arrays (data elements consecutive in memory) can be
 |      resized.
 |
 |      The purpose of the reference count check is to make sure you
 |      do not use this array as a buffer for another Python object and then
 |      reallocate the memory. However, reference counts can increase in
 |      other ways so if you are sure that you have not shared the memory
 |      for this array with another Python object, then you may safely set
 |      `refcheck` to False.
 |
 |      Examples
 |      --------
 |      Shrinking an array: array is flattened (in the order that the data a
re
 |      stored in memory), resized, and reshaped:
 |
 |      >>> a = np.array([[0, 1], [2, 3]], order='C')
 |      >>> a.resize((2, 1))
 |      >>> a
 |      array([[0],
 |             [1]])
 |
 |      >>> a = np.array([[0, 1], [2, 3]], order='F')
 |      >>> a.resize((2, 1))
```

Loading [MathJax]/extensions/Safe.js

```
 |      >>> a
 |      array([[0],
 |             [2]])
 |
 |      Enlarging an array: as above, but missing entries are filled with ze
ros:
 |
 |      >>> b = np.array([[0, 1], [2, 3]])
 |      >>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
 |      >>> b
 |      array([[0, 1, 2],
 |             [3, 0, 0]])
 |
 |      Referencing an array prevents resizing...
 |
 |      >>> c = a
 |      >>> a.resize((1, 1))
 |      Traceback (most recent call last):
 |      ...
 |      ValueError: cannot resize an array that references or is referenced
...
 |
 |      Unless `refcheck` is False:
 |
 |      >>> a.resize((1, 1), refcheck=False)
 |      >>> a
 |      array([[0]])
 |      >>> c
 |      array([[0]])
 |
 |  round(...)
 |      a.round(decimals=0, out=None)
 |
 |      Return `a` with each element rounded to the given number of decimal
s.
 |
 |      Refer to `numpy.around` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.around : equivalent function
 |
 |  searchsorted(...)
 |      a.searchsorted(v, side='left', sorter=None)
 |
 |      Find indices where elements of v should be inserted in a to maintain
order.
 |
 |      For full documentation, see `numpy.searchsorted`
 |
 |      See Also
 |      --------
 |      numpy.searchsorted : equivalent function
 |
 |  setfield(...)
 |      a.setfield(val, dtype, offset=0)
```

Loading [MathJax]/extensions/Safe.js

```
 |
 |      Put a value into a specified place in a field defined by a data-typ
e.
 |
 |      Place `val` into `a`'s field defined by `dtype` and beginning `offse
t`
 |      bytes into the field.
 |
 |      Parameters
 |      ----------
 |      val : object
 |          Value to be placed in field.
 |      dtype : dtype object
 |          Data-type of the field in which to place `val`.
 |      offset : int, optional
 |          The number of bytes into the field at which to place `val`.
 |
 |      Returns
 |      -------
 |      None
 |
 |      See Also
 |      --------
 |      getfield
 |
 |      Examples
 |      --------
 |      >>> x = np.eye(3)
 |      >>> x.getfield(np.float64)
 |      array([[1.,  0.,  0.],
 |             [0.,  1.,  0.],
 |             [0.,  0.,  1.]])
 |      >>> x.setfield(3, np.int32)
 |      >>> x.getfield(np.int32)
 |      array([[3, 3, 3],
 |             [3, 3, 3],
 |             [3, 3, 3]], dtype=int32)
 |      >>> x
 |      array([[1.0e+000, 1.5e-323, 1.5e-323],
 |             [1.5e-323, 1.0e+000, 1.5e-323],
 |             [1.5e-323, 1.5e-323, 1.0e+000]])
 |      >>> x.setfield(np.eye(3), np.int32)
 |      >>> x
 |      array([[1.,  0.,  0.],
 |             [0.,  1.,  0.],
 |             [0.,  0.,  1.]])
 |
 |  setflags(...)
 |      a.setflags(write=None, align=None, uic=None)
 |
 |      Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY,
 |      respectively.
 |
 |      These Boolean-valued flags affect how numpy interprets the memory
 |      area used by `a` (see Notes below). The ALIGNED flag can only
 |      be set to True if the data is actually aligned according to the typ
```

Loading [MathJax]/extensions/Safe.js

```
e.
 |      The WRITEBACKIFCOPY and flag can never be set
 |      to True. The flag WRITEABLE can only be set to True if the array own
s its
 |      own memory, or the ultimate owner of the memory exposes a writeable
buffer
 |      interface, or is a string. (The exception for string is made so that
 |      unpickling can be done without copying memory.)
 |
 |      Parameters
 |      ----------
 |      write : bool, optional
 |          Describes whether or not `a` can be written to.
 |      align : bool, optional
 |          Describes whether or not `a` is aligned properly for its type.
 |      uic : bool, optional
 |          Describes whether or not `a` is a copy of another "base" array.
 |
 |      Notes
 |      -----
 |      Array flags provide information about how the memory area used
 |      for the array is to be interpreted. There are 7 Boolean flags
 |      in use, only four of which can be changed by the user:
 |      WRITEBACKIFCOPY, WRITEABLE, and ALIGNED.
 |
 |      WRITEABLE (W) the data area can be written to;
 |
 |      ALIGNED (A) the data and strides are aligned appropriately for the h
ardware
 |      (as determined by the compiler);
 |
 |      WRITEBACKIFCOPY (X) this array is a copy of some other array (refere
nced
 |      by .base). When the C-API function PyArray_ResolveWritebackIfCopy is
 |      called, the base array will be updated with the contents of this arr
ay.
 |
 |      All flags can be accessed using the single (upper case) letter as we
ll
 |      as the full name.
 |
 |      Examples
 |      --------
 |      >>> y = np.array([[3, 1, 7],
 |      ...               [2, 0, 0],
 |      ...               [8, 5, 9]])
 |      >>> y
 |      array([[3, 1, 7],
 |             [2, 0, 0],
 |             [8, 5, 9]])
 |      >>> y.flags
 |        C_CONTIGUOUS : True
 |        F_CONTIGUOUS : False
 |        OWNDATA : True
 |        WRITEABLE : True
 |        ALIGNED : True
```

Loading [MathJax]/extensions/Safe.js

```
 |       WRITEBACKIFCOPY : False
 |     >>> y.setflags(write=0, align=0)
 |     >>> y.flags
 |       C_CONTIGUOUS : True
 |       F_CONTIGUOUS : False
 |       OWNDATA : True
 |       WRITEABLE : False
 |       ALIGNED : False
 |       WRITEBACKIFCOPY : False
 |     >>> y.setflags(uic=1)
 |     Traceback (most recent call last):
 |       File "<stdin>", line 1, in <module>
 |     ValueError: cannot set WRITEBACKIFCOPY flag to True
 |
 |  sort(...)
 |      a.sort(axis=-1, kind=None, order=None)
 |
 |      Sort an array in-place. Refer to `numpy.sort` for full documentatio
n.
 |
 |      Parameters
 |      ----------
 |      axis : int, optional
 |          Axis along which to sort. Default is -1, which means sort along
the
 |          last axis.
 |      kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
 |          Sorting algorithm. The default is 'quicksort'. Note that both 's
table'
 |          and 'mergesort' use timsort under the covers and, in general, th
e
 |          actual implementation will vary with datatype. The 'mergesort' o
ption
 |          is retained for backwards compatibility.
 |
 |          .. versionchanged:: 1.15.0
 |             The 'stable' option was added.
 |
 |      order : str or list of str, optional
 |          When `a` is an array with fields defined, this argument specifie
s
 |          which fields to compare first, second, etc.  A single field can
 |          be specified as a string, and not all fields need be specified,
 |          but unspecified fields will still be used, in the order in which
 |          they come up in the dtype, to break ties.
 |
 |      See Also
 |      --------
 |      numpy.sort : Return a sorted copy of an array.
 |      numpy.argsort : Indirect sort.
 |      numpy.lexsort : Indirect stable sort on multiple keys.
 |      numpy.searchsorted : Find elements in sorted array.
 |      numpy.partition: Partial sort.
 |
 |      Notes
 |      -----
```

Loading [MathJax]/extensions/Safe.js

```
 |      See `numpy.sort` for notes on the different sorting algorithms.
 |
 |      Examples
 |      --------
 |      >>> a = np.array([[1,4], [3,1]])
 |      >>> a.sort(axis=1)
 |      >>> a
 |      array([[1, 4],
 |             [1, 3]])
 |      >>> a.sort(axis=0)
 |      >>> a
 |      array([[1, 3],
 |             [1, 4]])
 |
 |      Use the `order` keyword to specify a field to use when sorting a
 |      structured array:
 |
 |      >>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', in
t)])
 |      >>> a.sort(order='y')
 |      >>> a
 |      array([(b'c', 1), (b'a', 2)],
 |            dtype=[('x', 'S1'), ('y', '<i8')])
 |
 |  squeeze(...)
 |      a.squeeze(axis=None)
 |
 |      Remove axes of length one from `a`.
 |
 |      Refer to `numpy.squeeze` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.squeeze : equivalent function
 |
 |  std(...)
 |      a.std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, wh
ere=True)
 |
 |      Returns the standard deviation of the array elements along given axi
s.
 |
 |      Refer to `numpy.std` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.std : equivalent function
 |
 |  sum(...)
 |      a.sum(axis=None, dtype=None, out=None, keepdims=False, initial=0, wh
ere=True)
 |
 |      Return the sum of the array elements over the given axis.
 |
 |      Refer to `numpy.sum` for full documentation.
 |
```

Loading [MathJax]/extensions/Safe.js

```
 |      See Also
 |      --------
 |      numpy.sum : equivalent function
 |
 |  swapaxes(...)
 |      a.swapaxes(axis1, axis2)
 |
 |      Return a view of the array with `axis1` and `axis2` interchanged.
 |
 |      Refer to `numpy.swapaxes` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.swapaxes : equivalent function
 |
 |  take(...)
 |      a.take(indices, axis=None, out=None, mode='raise')
 |
 |      Return an array formed from the elements of `a` at the given indice
s.
 |
 |      Refer to `numpy.take` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.take : equivalent function
 |
 |  tobytes(...)
 |      a.tobytes(order='C')
 |
 |      Construct Python bytes containing the raw data bytes in the array.
 |
 |      Constructs Python bytes showing a copy of the raw contents of
 |      data memory. The bytes object is produced in C-order by default.
 |      This behavior is controlled by the ``order`` parameter.
 |
 |      .. versionadded:: 1.9.0
 |
 |      Parameters
 |      ----------
 |      order : {'C', 'F', 'A'}, optional
 |          Controls the memory layout of the bytes object. 'C' means C-orde
r,
 |          'F' means F-order, 'A' (short for *Any*) means 'F' if `a` is
 |          Fortran contiguous, 'C' otherwise. Default is 'C'.
 |
 |      Returns
 |      -------
 |      s : bytes
 |          Python bytes exhibiting a copy of `a`'s raw data.
 |
 |      See also
 |      --------
 |      frombuffer
 |          Inverse of this operation, construct a 1-dimensional array from
Python
```

Loading [MathJax]/extensions/Safe.js

```
 |          bytes.
 |
 |      Examples
 |      --------
 |      >>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
 |      >>> x.tobytes()
 |      b'\x00\x00\x01\x00\x02\x00\x03\x00'
 |      >>> x.tobytes('C') == x.tobytes()
 |      True
 |      >>> x.tobytes('F')
 |      b'\x00\x00\x02\x00\x01\x00\x03\x00'
 |
 |  tofile(...)
 |      a.tofile(fid, sep="", format="%s")
 |
 |      Write array to a file as text or binary (default).
 |
 |      Data is always written in 'C' order, independent of the order of `a
`.
 |      The data produced by this method can be recovered using the function
 |      fromfile().
 |
 |      Parameters
 |      ----------
 |      fid : file or str or Path
 |          An open file object, or a string containing a filename.
 |
 |          .. versionchanged:: 1.17.0
 |              `pathlib.Path` objects are now accepted.
 |
 |      sep : str
 |          Separator between array items for text output.
 |          If "" (empty), a binary file is written, equivalent to
 |          ``file.write(a.tobytes())``.
 |      format : str
 |          Format string for text file output.
 |          Each entry in the array is formatted to text by first converting
 |          it to the closest Python type, and then using "format" % item.
 |
 |      Notes
 |      -----
 |      This is a convenience function for quick storage of array data.
 |      Information on endianness and precision is lost, so this method is n
ot a
 |      good choice for files intended to archive data or transport data bet
ween
 |      machines with different endianness. Some of these problems can be ov
ercome
 |      by outputting the data as text files, at the expense of speed and fi
le
 |      size.
 |
 |      When fid is a file object, array contents are directly written to th
e
 |      file, bypassing the file object's ``write`` method. As a result, tof
ile
```

Loading [MathJax]/extensions/Safe.js

```
 |      cannot be used with files objects supporting compression (e.g., Gzip
File)
 |      or file-like objects that do not support ``fileno()`` (e.g., BytesI
O).
 |
 |  tolist(...)
 |      a.tolist()
 |
 |      Return the array as an ``a.ndim``-levels deep nested list of Python
scalars.
 |
 |      Return a copy of the array data as a (nested) Python list.
 |      Data items are converted to the nearest compatible builtin Python ty
pe, via
 |      the `~numpy.ndarray.item` function.
 |
 |      If ``a.ndim`` is 0, then since the depth of the nested list is 0, it
will
 |      not be a list at all, but a simple Python scalar.
 |
 |      Parameters
 |      ----------
 |      none
 |
 |      Returns
 |      -------
 |      y : object, or list of object, or list of list of object, or ...
 |          The possibly nested list of array elements.
 |
 |      Notes
 |      -----
 |      The array may be recreated via ``a = np.array(a.tolist())``, althoug
h this
 |      may sometimes lose precision.
 |
 |      Examples
 |      --------
 |      For a 1D array, ``a.tolist()`` is almost the same as ``list(a)``,
 |      except that ``tolist`` changes numpy scalars to Python scalars:
 |
 |      >>> a = np.uint32([1, 2])
 |      >>> a_list = list(a)
 |      >>> a_list
 |      [1, 2]
 |      >>> type(a_list[0])
 |      <class 'numpy.uint32'>
 |      >>> a_tolist = a.tolist()
 |      >>> a_tolist
 |      [1, 2]
 |      >>> type(a_tolist[0])
 |      <class 'int'>
 |
 |      Additionally, for a 2D array, ``tolist`` applies recursively:
 |
 |      >>> a = np.array([[1, 2], [3, 4]])
 |      >>> list(a)
```

Loading [MathJax]/extensions/Safe.js

```
|      [array([1, 2]), array([3, 4])]
|      >>> a.tolist()
|      [[1, 2], [3, 4]]
|
|      The base case for this recursion is a 0D array:
|
|      >>> a = np.array(1)
|      >>> list(a)
|      Traceback (most recent call last):
|        ...
|      TypeError: iteration over a 0-d array
|      >>> a.tolist()
|      1
|
|  tostring(...)
|      a.tostring(order='C')
|
|      A compatibility alias for `tobytes`, with exactly the same behavior.
|
|      Despite its name, it returns `bytes` not `str`\ s.
|
|      .. deprecated:: 1.19.0
|
|  trace(...)
|      a.trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)
|
|      Return the sum along diagonals of the array.
|
|      Refer to `numpy.trace` for full documentation.
|
|      See Also
|      --------
|      numpy.trace : equivalent function
|
|  transpose(...)
|      a.transpose(*axes)
|
|      Returns a view of the array with axes transposed.
|
|      Refer to `numpy.transpose` for full documentation.
|
|      Parameters
|      ----------
|      axes : None, tuple of ints, or `n` ints
|
|       * None or no argument: reverses the order of the axes.
|
|       * tuple of ints: `i` in the `j`-th place in the tuple means that th
e
|         array's `i`-th axis becomes the transposed array's `j`-th axis.
|
|       * `n` ints: same as an n-tuple of the same ints (this form is
|         intended simply as a "convenience" alternative to the tuple for
m).
|
|      Returns
```

Loading [MathJax]/extensions/Safe.js

```
 |      -------
 |      p : ndarray
 |          View of the array with its axes suitably permuted.
 |
 |      See Also
 |      --------
 |      transpose : Equivalent function.
 |      ndarray.T : Array property returning the array transposed.
 |      ndarray.reshape : Give a new shape to an array without changing its
data.
 |
 |      Examples
 |      --------
 |      >>> a = np.array([[1, 2], [3, 4]])
 |      >>> a
 |      array([[1, 2],
 |             [3, 4]])
 |      >>> a.transpose()
 |      array([[1, 3],
 |             [2, 4]])
 |      >>> a.transpose((1, 0))
 |      array([[1, 3],
 |             [2, 4]])
 |      >>> a.transpose(1, 0)
 |      array([[1, 3],
 |             [2, 4]])
 |
 |      >>> a = np.array([1, 2, 3, 4])
 |      >>> a
 |      array([1, 2, 3, 4])
 |      >>> a.transpose()
 |      array([1, 2, 3, 4])
 |
 |  var(...)
 |      a.var(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, wh
ere=True)
 |
 |      Returns the variance of the array elements, along given axis.
 |
 |      Refer to `numpy.var` for full documentation.
 |
 |      See Also
 |      --------
 |      numpy.var : equivalent function
 |
 |  view(...)
 |      a.view([dtype][, type])
 |
 |      New view of array with the same data.
 |
 |      .. note::
 |          Passing None for ``dtype`` is different from omitting the parame
ter,
 |          since the former invokes ``dtype(None)`` which is an alias for
 |          ``dtype('float_')``.
```

Loading [MathJax]/extensions/Safe.js

```
 |      Parameters
 |      ----------
 |      dtype : data-type or ndarray sub-class, optional
 |          Data-type descriptor of the returned view, e.g., float32 or int1
6.
 |          Omitting it results in the view having the same data-type as `a
`.
 |          This argument can also be specified as an ndarray sub-class, whi
ch
 |          then specifies the type of the returned object (this is equivale
nt to
 |          setting the ``type`` parameter).
 |      type : Python type, optional
 |          Type of the returned view, e.g., ndarray or matrix.  Again, omis
sion
 |          of the parameter results in type preservation.
 |
 |      Notes
 |      -----
 |      ``a.view()`` is used two different ways:
 |
 |      ``a.view(some_dtype)`` or ``a.view(dtype=some_dtype)`` constructs a
view
 |      of the array's memory with a different data-type.  This can cause a
 |      reinterpretation of the bytes of memory.
 |
 |      ``a.view(ndarray_subclass)`` or ``a.view(type=ndarray_subclass)`` ju
st
 |      returns an instance of `ndarray_subclass` that looks at the same arr
ay
 |      (same shape, dtype, etc.)  This does not cause a reinterpretation of
the
 |      memory.
 |
 |      For ``a.view(some_dtype)``, if ``some_dtype`` has a different number
of
 |      bytes per entry than the previous dtype (for example, converting a r
egular
 |      array to a structured array), then the last axis of ``a`` must be
 |      contiguous. This axis will be resized in the result.
 |
 |      .. versionchanged:: 1.23.0
 |         Only the last axis needs to be contiguous. Previously, the entire
array
 |         had to be C-contiguous.
 |
 |      Examples
 |      --------
 |      >>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
 |
 |      Viewing array data using a different type and dtype:
 |
 |      >>> y = x.view(dtype=np.int16, type=np.matrix)
 |      >>> y
 |      matrix([[513]], dtype=int16)
 |      >>> print(type(y))
```

Loading [MathJax]/extensions/Safe.js

```
 |      <class 'numpy.matrix'>
 |
 |      Creating a view on a structured array so it can be used in calculati
ons
 |
 |      >>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int
8)])
 |      >>> xv = x.view(dtype=np.int8).reshape(-1,2)
 |      >>> xv
 |      array([[1, 2],
 |             [3, 4]], dtype=int8)
 |      >>> xv.mean(0)
 |      array([2.,  3.])
 |
 |      Making changes to the view changes the underlying array
 |
 |      >>> xv[0,1] = 20
 |      >>> x
 |      array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
 |
 |      Using a view to convert an array to a recarray:
 |
 |      >>> z = x.view(np.recarray)
 |      >>> z.a
 |      array([1, 3], dtype=int8)
 |
 |      Views share data:
 |
 |      >>> x[0] = (9, 10)
 |      >>> z[0]
 |      (9, 10)
 |
 |      Views that change the dtype size (bytes per entry) should normally b
e
 |      avoided on arrays defined by slices, transposes, fortran-ordering, e
tc.:
 |
 |      >>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
 |      >>> y = x[:, ::2]
 |      >>> y
 |      array([[1, 3],
 |             [4, 6]], dtype=int16)
 |      >>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
 |      Traceback (most recent call last):
 |          ...
 |      ValueError: To change to a dtype of a different size, the last axis
must be contiguous
 |      >>> z = y.copy()
 |      >>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
 |      array([[(1, 3)],
 |             [(4, 6)]], dtype=[('width', '<i2'), ('length', '<i2')])
 |
 |      However, views that change dtype are totally fine for arrays with a
 |      contiguous last axis, even if the rest of the axes are not C-contigu
```

Loading [MathJax]/extensions/Safe.js

```
|      >>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
|      >>> x.transpose(1, 0, 2).view(np.int16)
|      array([[[ 256,  770],
|              [3340, 3854]],
|      <BLANKLINE>
|             [[1284, 1798],
|              [4368, 4882]],
|      <BLANKLINE>
|             [[2312, 2826],
|              [5396, 5910]]], dtype=int16)
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  __class_getitem__(...) from builtins.type
|      a.__class_getitem__(item, /)
|
|      Return a parametrized wrapper around the `~numpy.ndarray` type.
|
|      .. versionadded:: 1.22
|
|      Returns
|      -------
|      alias : types.GenericAlias
|          A parametrized `~numpy.ndarray` type.
|
|      Examples
|      --------
|      >>> from typing import Any
|      >>> import numpy as np
|
|      >>> np.ndarray[Any, np.dtype[Any]]
|      numpy.ndarray[typing.Any, numpy.dtype[typing.Any]]
|
|      See Also
|      --------
|      :pep:`585` : Type hinting generics in standard collections.
|      numpy.typing.NDArray : An ndarray alias :term:`generic <generic type
>`
|                          w.r.t. its `dtype.type <numpy.dtype.type>`.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signatu
re.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  T
|      View of the transposed array.
|
|      Same as ``self.transpose()``.
|
```

Loading [MathJax]/extensions/Safe.js

```
|      Examples
|      --------
|      >>> a = np.array([[1, 2], [3, 4]])
|      >>> a
|      array([[1, 2],
|             [3, 4]])
|      >>> a.T
|      array([[1, 3],
|             [2, 4]])
|
|      >>> a = np.array([1, 2, 3, 4])
|      >>> a
|      array([1, 2, 3, 4])
|      >>> a.T
|      array([1, 2, 3, 4])
|
|      See Also
|      --------
|      transpose
|
|  __array_interface__
|      Array protocol: Python side.
|
|  __array_priority__
|      Array priority.
|
|  __array_struct__
|      Array protocol: C-struct side.
|
|  base
|      Base object if memory is from some other object.
|
|      Examples
|      --------
|      The base of an array that owns its memory is None:
|
|      >>> x = np.array([1,2,3,4])
|      >>> x.base is None
|      True
|
|      Slicing creates a view, whose memory is shared with x:
|
|      >>> y = x[2:]
|      >>> y.base is x
|      True
|
|  ctypes
|      An object to simplify the interaction of the array with the ctypes
|      module.
|
|      This attribute creates an object that makes it easier to use arrays
|      when calling shared libraries with the ctypes module. The returned
|      object has, among others, data, shape, and strides attributes (see
|      Notes below) which themselves return ctypes objects that can be used
|      as arguments to a shared library.
|
```

Loading [MathJax]/extensions/Safe.js

```
|      Parameters
|      ----------
|      None
|
|      Returns
|      -------
|      c : Python object
|          Possessing attributes data, shape, strides, etc.
|
|      See Also
|      --------
|      numpy.ctypeslib
|
|      Notes
|      -----
|      Below are the public attributes of this object which were documented
|      in "Guide to NumPy" (we have omitted undocumented public attributes,
|      as well as documented private attributes):
|
|      .. autoattribute:: numpy.core._internal._ctypes.data
|          :noindex:
|
|      .. autoattribute:: numpy.core._internal._ctypes.shape
|          :noindex:
|
|      .. autoattribute:: numpy.core._internal._ctypes.strides
|          :noindex:
|
|      .. automethod:: numpy.core._internal._ctypes.data_as
|          :noindex:
|
|      .. automethod:: numpy.core._internal._ctypes.shape_as
|          :noindex:
|
|      .. automethod:: numpy.core._internal._ctypes.strides_as
|          :noindex:
|
|      If the ctypes module is not available, then the ctypes attribute
|      of array objects still returns something useful, but ctypes objects
|      are not returned and errors may be raised instead. In particular,
|      the object will still have the ``as_parameter`` attribute which will
|      return an integer equal to the data attribute.
|
|      Examples
|      --------
|      >>> import ctypes
|      >>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
|      >>> x
|      array([[0, 1],
|             [2, 3]], dtype=int32)
|      >>> x.ctypes.data
|      31962608 # may vary
|      >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
|      <__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
|      >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
|      c_uint(0)
```

Loading [MathJax]/extensions/Safe.js

```
 |      >>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
 |      c_ulong(4294967296)
 |      >>> x.ctypes.shape
 |      <numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may
vary
 |      >>> x.ctypes.strides
 |      <numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may
vary
 |
 |  data
 |      Python buffer object pointing to the start of the array's data.
 |
 |  dtype
 |      Data-type of the array's elements.
 |
 |      .. warning::
 |
 |          Setting ``arr.dtype`` is discouraged and may be deprecated in th
e
 |          future.  Setting will replace the ``dtype`` without modifying th
e
 |          memory (see also `ndarray.view` and `ndarray.astype`).
 |
 |      Parameters
 |      ----------
 |      None
 |
 |      Returns
 |      -------
 |      d : numpy dtype object
 |
 |      See Also
 |      --------
 |      ndarray.astype : Cast the values contained in the array to a new dat
a-type.
 |      ndarray.view : Create a view of the same data but a different data-t
ype.
 |      numpy.dtype
 |
 |      Examples
 |      --------
 |      >>> x
 |      array([[0, 1],
 |             [2, 3]])
 |      >>> x.dtype
 |      dtype('int32')
 |      >>> type(x.dtype)
 |      <type 'numpy.dtype'>
 |
 |  flags
 |      Information about the memory layout of the array.
 |
 |      Attributes
 |      ----------
 |      C_CONTIGUOUS (C)
 |          The data is in a single, C-style contiguous segment.
```

Loading [MathJax]/extensions/Safe.js

```
 |      F_CONTIGUOUS (F)
 |          The data is in a single, Fortran-style contiguous segment.
 |      OWNDATA (O)
 |          The array owns the memory it uses or borrows it from another obj
ect.
 |      WRITEABLE (W)
 |          The data area can be written to.  Setting this to False locks
 |          the data, making it read-only.  A view (slice, etc.) inherits WR
ITEABLE
 |          from its base array at creation time, but a view of a writeable
 |          array may be subsequently locked while the base array remains wr
iteable.
 |          (The opposite is not true, in that a view of a locked array may
not
 |          be made writeable.  However, currently, locking a base object do
es not
 |          lock any views that already reference it, so under that circumst
ance it
 |          is possible to alter the contents of a locked array via a previo
usly
 |          created writeable view onto it.)  Attempting to change a non-wri
teable
 |          array raises a RuntimeError exception.
 |      ALIGNED (A)
 |          The data and all elements are aligned appropriately for the hard
ware.
 |      WRITEBACKIFCOPY (X)
 |          This array is a copy of some other array. The C-API function
 |          PyArray_ResolveWritebackIfCopy must be called before deallocatin
g
 |          to the base array will be updated with the contents of this arra
y.
 |      FNC
 |          F_CONTIGUOUS and not C_CONTIGUOUS.
 |      FORC
 |          F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
 |      BEHAVED (B)
 |          ALIGNED and WRITEABLE.
 |      CARRAY (CA)
 |          BEHAVED and C_CONTIGUOUS.
 |      FARRAY (FA)
 |          BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.
 |
 |      Notes
 |      -----
 |      The `flags` object can be accessed dictionary-like (as in ``a.flags
['WRITEABLE']``),
 |      or by using lowercased attribute names (as in ``a.flags.writeable`
`). Short flag
 |      names are only supported in dictionary access.
 |
 |      Only the WRITEBACKIFCOPY, WRITEABLE, and ALIGNED flags can be
 |      changed by the user, via direct assignment to the attribute or dicti
onary
 |      entry, or by calling `ndarray.setflags`.
 |
```

Loading [MathJax]/extensions/Safe.js

```
 |      The array flags cannot be set arbitrarily:
 |
 |      - WRITEBACKIFCOPY can only be set ``False``.
 |      - ALIGNED can only be set ``True`` if the data is truly aligned.
 |      - WRITEABLE can only be set ``True`` if the array owns its own memor
y
 |        or the ultimate owner of the memory exposes a writeable buffer
 |        interface or is a string.
 |
 |      Arrays can be both C-style and Fortran-style contiguous simultaneous
ly.
 |      This is clear for 1-dimensional arrays, but can also be true for hig
her
 |      dimensional arrays.
 |
 |      Even for contiguous arrays a stride for a given dimension
 |      ``arr.strides[dim]`` may be *arbitrary* if ``arr.shape[dim] == 1``
 |      or the array has no elements.
 |      It does *not* generally hold that ``self.strides[-1] == self.itemsiz
e``
 |      for C-style contiguous arrays or ``self.strides[0] == self.itemsize`
` for
 |      Fortran-style contiguous arrays is true.
 |
 |  flat
 |      A 1-D iterator over the array.
 |
 |      This is a `numpy.flatiter` instance, which acts similarly to, but is
not
 |      a subclass of, Python's built-in iterator object.
 |
 |      See Also
 |      --------
 |      flatten : Return a copy of the array collapsed into one dimension.
 |
 |      flatiter
 |
 |      Examples
 |      --------
 |      >>> x = np.arange(1, 7).reshape(2, 3)
 |      >>> x
 |      array([[1, 2, 3],
 |             [4, 5, 6]])
 |      >>> x.flat[3]
 |      4
 |      >>> x.T
 |      array([[1, 4],
 |             [2, 5],
 |             [3, 6]])
 |      >>> x.T.flat[3]
 |      5
 |      >>> type(x.flat)
 |      <class 'numpy.flatiter'>
 |
 |      An assignment example:
 |
```

Loading [MathJax]/extensions/Safe.js

```
|      >>> x.flat = 3; x
|      array([[3, 3, 3],
|             [3, 3, 3]])
|      >>> x.flat[[1,4]] = 1; x
|      array([[3, 1, 3],
|             [3, 1, 3]])
|
|  imag
|      The imaginary part of the array.
|
|      Examples
|      --------
|      >>> x = np.sqrt([1+0j, 0+1j])
|      >>> x.imag
|      array([ 0.        ,  0.70710678])
|      >>> x.imag.dtype
|      dtype('float64')
|
|  itemsize
|      Length of one array element in bytes.
|
|      Examples
|      --------
|      >>> x = np.array([1,2,3], dtype=np.float64)
|      >>> x.itemsize
|      8
|      >>> x = np.array([1,2,3], dtype=np.complex128)
|      >>> x.itemsize
|      16
|
|  nbytes
|      Total bytes consumed by the elements of the array.
|
|      Notes
|      -----
|      Does not include memory consumed by non-element attributes of the
|      array object.
|
|      See Also
|      --------
|      sys.getsizeof
|          Memory consumed by the object itself without parents in case vie
w.
|          This does include memory consumed by non-element attributes.
|
|      Examples
|      --------
|      >>> x = np.zeros((3,5,2), dtype=np.complex128)
|      >>> x.nbytes
|      480
|      >>> np.prod(x.shape) * x.itemsize
|      480
|
|  ndim
|      Number of array dimensions.
|
```

Loading [MathJax]/extensions/Safe.js

```
 |      Examples
 |      --------
 |      >>> x = np.array([1, 2, 3])
 |      >>> x.ndim
 |      1
 |      >>> y = np.zeros((2, 3, 4))
 |      >>> y.ndim
 |      3
 |
 |  real
 |      The real part of the array.
 |
 |      Examples
 |      --------
 |      >>> x = np.sqrt([1+0j, 0+1j])
 |      >>> x.real
 |      array([ 1.        ,  0.70710678])
 |      >>> x.real.dtype
 |      dtype('float64')
 |
 |      See Also
 |      --------
 |      numpy.real : equivalent function
 |
 |  shape
 |      Tuple of array dimensions.
 |
 |      The shape property is usually used to get the current shape of an ar
ray,
 |      but may also be used to reshape the array in-place by assigning a tu
ple of
 |      array dimensions to it.  As with `numpy.reshape`, one of the new sha
pe
 |      dimensions can be -1, in which case its value is inferred from the s
ize of
 |      the array and the remaining dimensions. Reshaping an array in-place
will
 |      fail if a copy is required.
 |
 |      .. warning::
 |
 |          Setting ``arr.shape`` is discouraged and may be deprecated in th
e
 |          future.  Using `ndarray.reshape` is the preferred approach.
 |
 |      Examples
 |      --------
 |      >>> x = np.array([1, 2, 3, 4])
 |      >>> x.shape
 |      (4,)
 |      >>> y = np.zeros((2, 3, 4))
 |      >>> y.shape
 |      (2, 3, 4)
 |      >>> y.shape = (3, 8)
 |      >>> y
 |      array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
```

Loading [MathJax]/extensions/Safe.js

```
|             [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
|             [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
|      >>> y.shape = (3, 6)
|      Traceback (most recent call last):
|        File "<stdin>", line 1, in <module>
|      ValueError: total size of new array must be unchanged
|      >>> np.zeros((4,2))[::2].shape = (-1,)
|      Traceback (most recent call last):
|        File "<stdin>", line 1, in <module>
|      AttributeError: Incompatible shape for in-place modification. Use
|      `.reshape()` to make a copy with the desired shape.
|
|      See Also
|      --------
|      numpy.shape : Equivalent getter function.
|      numpy.reshape : Function similar to setting ``shape``.
|      ndarray.reshape : Method similar to setting ``shape``.
|
|  size
|      Number of elements in the array.
|
|      Equal to ``np.prod(a.shape)``, i.e., the product of the array's
|      dimensions.
|
|      Notes
|      -----
|      `a.size` returns a standard arbitrary precision Python integer. This
|      may not be the case with other methods of obtaining the same value
|      (like the suggested ``np.prod(a.shape)``, which returns an instance
|      of ``np.int_``), and may be relevant if the value is used further in
|      calculations that may overflow a fixed size integer type.
|
|      Examples
|      --------
|      >>> x = np.zeros((3, 5, 2), dtype=np.complex128)
|      >>> x.size
|      30
|      >>> np.prod(x.shape)
|      30
|
|  strides
|      Tuple of bytes to step in each dimension when traversing an array.
|
|      The byte offset of element ``(i[0], i[1], ..., i[n])`` in an array `
a`
|      is::
|
|          offset = sum(np.array(i) * a.strides)
|
|      A more detailed explanation of strides can be found in the
|      "ndarray.rst" file in the NumPy reference guide.
|
|      .. warning::
|
|          Setting ``arr.strides`` is discouraged and may be deprecated in
|      the
```

Loading [MathJax]/extensions/Safe.js

```
 |          future.  `numpy.lib.stride_tricks.as_strided` should be preferre
d
 |          to create a new view of the same data in a safer way.
 |
 |      Notes
 |      -----
 |      Imagine an array of 32-bit integers (each 4 bytes)::
 |
 |        x = np.array([[0, 1, 2, 3, 4],
 |                      [5, 6, 7, 8, 9]], dtype=np.int32)
 |
 |      This array is stored in memory as 40 bytes, one after the other
 |      (known as a contiguous block of memory).  The strides of an array te
ll
 |      us how many bytes we have to skip in memory to move to the next posi
tion
 |      along a certain axis.  For example, we have to skip 4 bytes (1 valu
e) to
 |      move to the next column, but 20 bytes (5 values) to get to the same
 |      position in the next row.  As such, the strides for the array `x` wi
ll be
 |      ``(20, 4)``.
 |
 |      See Also
 |      --------
 |      numpy.lib.stride_tricks.as_strided
 |
 |      Examples
 |      --------
 |      >>> y = np.reshape(np.arange(2*3*4), (2,3,4))
 |      >>> y
 |      array([[[ 0,  1,  2,  3],
 |              [ 4,  5,  6,  7],
 |              [ 8,  9, 10, 11]],
 |             [[12, 13, 14, 15],
 |              [16, 17, 18, 19],
 |              [20, 21, 22, 23]]])
 |      >>> y.strides
 |      (48, 16, 4)
 |      >>> y[1,1,1]
 |      17
 |      >>> offset=sum(y.strides * np.array((1,1,1)))
 |      >>> offset/y.itemsize
 |      17
 |
 |      >>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
 |      >>> x.strides
 |      (32, 4, 224, 1344)
 |      >>> i = np.array([3,5,2,2])
 |      >>> offset = sum(i * x.strides)
 |      >>> x[3,5,2,2]
 |      813
 |      >>> offset / x.itemsize
 |      813
 |
 |  ----------------------------------------------------------------------
```

Loading [MathJax]/extensions/Safe.js

```
|  Data and other attributes defined here:
|
|  __hash__ = None
```

(Of course, you might also prefer to check out the online docs.)

Operator overloading

What's the value of the below expression?

In [20]:
```
[3, 4, 1, 2, 2, 1] + 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 [3, 4, 1, 2, 2, 1] + 10

TypeError: can only concatenate list (not "int") to list
```

What a silly question. Of course it's an error.

But what about...

In [21]:
```
rolls + 10
```

Out[21]:
```
array([11, 11, 14, 13, 12, 11, 15, 11, 12, 12])
```

We might think that Python strictly polices how pieces of its core syntax behave such as `+`, `<`, `in`, `==`, or square brackets for indexing and slicing. But in fact, it takes a very hands-off approach. When you define a new type, you can choose how addition works for it, or what it means for an object of that type to be equal to something else.

The designers of lists decided that adding them to numbers wasn't allowed. The designers of `numpy` arrays went a different way (adding the number to each element of the array).

Here are a few more examples of how `numpy` arrays interact unexpectedly with Python operators (or at least differently from lists).

In [22]:
```
# At which indices are the dice less than or equal to 3?
rolls <= 3
```

Out[22]:
```
array([ True,  True, False,  True,  True,  True, False,  True,  True,
        True])
```

In [23]:
```
xlist = [[1,2,3],[2,4,6],]
# Create a 2-dimensional array
x = numpy.asarray(xlist)
print("xlist = {}\nx =\n{}".format(xlist, x))
```

Loading [MathJax]/extensions/Safe.js

```
xlist = [[1, 2, 3], [2, 4, 6]]
x =
[[1 2 3]
 [2 4 6]]
```

In [24]:
```
# Get the last element of the second row of our numpy array
x[1,-1]
```

Out[24]: 6

In [25]:
```
# Get the last element of the second sublist of our nested list?
xlist[1,-1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 2
      1 # Get the last element of the second sublist of our nested list?
----> 2 xlist[1,-1]

TypeError: list indices must be integers or slices, not tuple
```

numpy's `ndarray` type is specialized for working with multi-dimensional data, so it defines its own logic for indexing, allowing us to index by a tuple to specify the index at each dimension.

When does 1 + 1 not equal 2?

Things can get weirder than this. You may have heard of (or even used) tensorflow, a Python library popularly used for deep learning. It makes extensive use of operator overloading.

In [26]:
```
import tensorflow as tf
# Create two constants, each with value 1
a = tf.constant(1)
b = tf.constant(1)
# Add them together to get...
a + b
```

```
2025-03-11 19:48:47.911759: I metal_plugin/src/device/metal_device.cc:1154] 
Metal device set to: Apple M4 Pro
2025-03-11 19:48:47.911781: I metal_plugin/src/device/metal_device.cc:296] s
ystemMemory: 24.00 GB
2025-03-11 19:48:47.911789: I metal_plugin/src/device/metal_device.cc:313] m
axCacheSize: 8.00 GB
2025-03-11 19:48:47.912030: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:305] Could not identify NUMA node of platform
GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA sup
port.
2025-03-11 19:48:47.912037: I tensorflow/core/common_runtime/pluggable_devic
e/pluggable_device_factory.cc:271] Created TensorFlow device (/job:localhos
t/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevic
e (device: 0, name: METAL, pci bus id: <undefined>)
```

Loading [MathJax]/extensions/Safe.js

Out[26]: `<tf.Tensor: shape=(), dtype=int32, numpy=2>`

`a + b` isn't 2, it is (to quote tensorflow's documentation)...

> a symbolic handle to one of the outputs of an `Operation`. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow `tf.Session`.

It's important just to be aware of the fact that this sort of thing is possible and that libraries will often use operator overloading in non-obvious or magical-seeming ways.

Understanding how Python's operators work when applied to ints, strings, and lists is no guarantee that you'll be able to immediately understand what they do when applied to a tensorflow `Tensor`, or a numpy `ndarray`, or a pandas `DataFrame`.

Once you've had a little taste of DataFrames, for example, an expression like the one below starts to look appealingly intuitive:

```
# Get the rows with population over 1m in South America
df[(df['population'] > 10**6) & (df['continent'] == 'South
America')]
```

But why does it work? The example above features something like **5** different overloaded operators. What's each of those operations doing? It can help to know the answer when things start going wrong.

Curious how it all works?

Have you ever called `help()` or `dir()` on an object and wondered what the heck all those names with the double-underscores were?

In [27]:
```
print(dir(list))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__ge
tattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd_
_', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__le
n__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '_
_sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'coun
t', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

This turns out to be directly related to operator overloading.

When Python programmers want to define how operators behave on their types, they do so by implementing methods with special names beginning and ending with 2 underscores such as `__lt__`, `__setattr__`, or `__contains__`. Generally, names that follow this double-underscore format have a special meaning to Python.

Loading [MathJax]/extensions/Safe.js

So, for example, the expression `x in [1, 2, 3]` is actually calling the list method `__contains__` behind-the-scenes. It's equivalent to (the much uglier) `[1, 2, 3].__contains__(x)`.

If you're curious to learn more, you can check out Python's official documentation, which describes many, many more of these special "underscores" methods.

We won't be defining our own types in these lessons (if only there was time!), but I hope you'll get to experience the joys of defining your own wonderful, weird types later down the road.

Your turn!

Head over to **the final coding exercise** for one more round of coding questions involving imports, working with unfamiliar objects, and, of course, more gambling.

Loading [MathJax]/extensions/Safe.js