OOP: Object georiënteerd programmeren

Objectgeoriënteerd programmeren (OOP: Object Oriented Programming) is meestal een van de grootste obstakels voor beginners wanneer ze voor het eerst Python beginnen te leren.

Er zijn heel veel tutorials en lessen over OOP, dus voel je vrij om op Google te zoeken naar andere lessen, en ik heb ook enkele links naar andere nuttige tutorials online onder aan dit notitieblok geplaatst.

Voor deze les bouwen we onze kennis van OOP in Python op door voort te bouwen op de volgende onderwerpen:

- Voorwerpen
- Gebruik het *class* trefwoord
- Klassenattributen maken
- Methoden maken in een klasse
- Leren over inheritance (erfelijkheid)
- Leren over polymorfisme
- Leren over speciale methoden voor lessen

Laten we de les beginnen door te onthouden over de basis python-objecten. Bijvoorbeeld:

```
lst = [1,2,3]
```

Weet je nog hoe we methoden op een lijst konden aanroepen?

```
lst.count(2)
```

```
1
```

Wat we in deze lezing gaan doen, is onderzoeken hoe we een objecttype zoals een lijst kunnen maken. We hebben al geleerd hoe u functies maakt. Laten we objecten in het algemeen onderzoeken:

Voorwerpen

In Python is *alles een object*. Onthoud uit eerdere lezingen dat we type() kunnen gebruiken om te controleren welk type object iets is:

```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
```

```
<class 'dict'>
```

Dus we weten dat al deze dingen objecten zijn, dus hoe kunnen we onze eigen objecttypes maken?

class

De gebruiker gedefinieerde (user-defined) objecten worden gemaakt met het trefwoord class. De klasse is een blauwdruk/sjabloon die de aard van een toekomstig object definieert. Van klassen kunnen we instanties construeren. Een instantie is een specifiek object dat is gemaakt op basis van een bepaalde klasse. Hierboven hebben we bijvoorbeeld het object lst gemaakt, dat een instantie was van een list-object.

Laten we eens kijken hoe we class kunnen gebruiken:

```
x = 10

print(type(x))
```

```
<class 'int'>
```

```
help(int)
```

```
Help on class int in module builtins:

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 literal.
 |  >>> 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.
|
|  __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
|
|  __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, /)
|      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, /)
|      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 original 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, byteorder, *, 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.
|      byteorder
|        The byte order used to represent the integer.  If byteorder is 'big',
|        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.
|      signed
|        Determines whether two's complement is used to represent the integer.
|        If signed is False and a negative integer is given, an OverflowError
|        is raised.
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  from_bytes(bytes, byteorder, *, 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 bytes.
|        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 'big',
|        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.
|      signed
|        Indicates whether two's complement is used to represent the integer.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  denominator
```

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

```
s = "Hello World"
```

```
print(type(s))
```

```
<class 'str'>
```

```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_spec.
 |
 |  __ge__(self, value, /)
```

```
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(...)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
```

```
|  __sizeof__(self, /)
|      Return the size of the string in memory, in bytes.
|
|  __str__(self, /)
|      Return str(self).
|
|  capitalize(self, /)
|      Return a capitalized version of the string.
|
|      More specifically, make the first character have upper case and the rest lower
|      case.
|
|  casefold(self, /)
|      Return a version of the string suitable for caseless comparisons.
|
|  center(self, width, fillchar=' ', /)
|      Return a centered string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  count(...)
|      S.count(sub[, start[, end]]) -> int
|
|      Return the number of non-overlapping occurrences of substring sub in
|      string S[start:end].  Optional arguments start and end are
|      interpreted as in slice notation.
|
|  encode(self, /, encoding='utf-8', errors='strict')
|      Encode the string using the codec registered for encoding.
|
|      encoding
|        The encoding in which to encode the string.
|      errors
|        The error handling scheme to use for encoding errors.
|        The default is 'strict' meaning that encoding errors raise a
|        UnicodeEncodeError.  Other possible values are 'ignore', 'replace' and
|        'xmlcharrefreplace' as well as any other name registered with
|        codecs.register_error that can handle UnicodeEncodeErrors.
|
|  endswith(...)
|      S.endswith(suffix[, start[, end]]) -> bool
|
|      Return True if S ends with the specified suffix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      suffix can also be a tuple of strings to try.
```

```
|
|  expandtabs(self, /, tabsize=8)
|      Return a copy where all tab characters are expanded using spaces.
|
|      If tabsize is not given, a tab size of 8 characters is assumed.
|
|  find(...)
|      S.find(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  format(...)
|      S.format(*args, **kwargs) -> str
|
|      Return a formatted version of S, using substitutions from args and kwargs.
|      The substitutions are identified by braces ('{' and '}').
|
|  format_map(...)
|      S.format_map(mapping) -> str
|
|      Return a formatted version of S, using substitutions from mapping.
|      The substitutions are identified by braces ('{' and '}').
|
|  index(...)
|      S.index(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  isalnum(self, /)
|      Return True if the string is an alpha-numeric string, False otherwise.
|
|      A string is alpha-numeric if all characters in the string are alpha-numeric and
|      there is at least one character in the string.
|
|  isalpha(self, /)
|      Return True if the string is an alphabetic string, False otherwise.
|
|      A string is alphabetic if all characters in the string are alphabetic and there
```

```
|      is at least one character in the string.
|
|  isascii(self, /)
|      Return True if all characters in the string are ASCII, False otherwise.
|
|      ASCII characters have code points in the range U+0000-U+007F.
|      Empty string is ASCII too.
|
|  isdecimal(self, /)
|      Return True if the string is a decimal string, False otherwise.
|
|      A string is a decimal string if all characters in the string are decimal and
|      there is at least one character in the string.
|
|  isdigit(self, /)
|      Return True if the string is a digit string, False otherwise.
|
|      A string is a digit string if all characters in the string are digits and there
|      is at least one character in the string.
|
|  isidentifier(self, /)
|      Return True if the string is a valid Python identifier, False otherwise.
|
|      Call keyword.iskeyword(s) to test whether string s is a reserved identifier,
|      such as "def" or "class".
|
|  islower(self, /)
|      Return True if the string is a lowercase string, False otherwise.
|
|      A string is lowercase if all cased characters in the string are lowercase and
|      there is at least one cased character in the string.
|
|  isnumeric(self, /)
|      Return True if the string is a numeric string, False otherwise.
|
|      A string is numeric if all characters in the string are numeric and there is at
|      least one character in the string.
|
|  isprintable(self, /)
|      Return True if the string is printable, False otherwise.
|
|      A string is printable if all of its characters are considered printable in
|      repr() or if it is empty.
|
|  isspace(self, /)
|      Return True if the string is a whitespace string, False otherwise.
```

```
|
|      A string is whitespace if all characters in the string are whitespace and there
|      is at least one character in the string.
|
|  istitle(self, /)
|      Return True if the string is a title-cased string, False otherwise.
|
|      In a title-cased string, upper- and title-case characters may only
|      follow uncased characters and lowercase characters only cased ones.
|
|  isupper(self, /)
|      Return True if the string is an uppercase string, False otherwise.
|
|      A string is uppercase if all cased characters in the string are uppercase and
|      there is at least one cased character in the string.
|
|  join(self, iterable, /)
|      Concatenate any number of strings.
|
|      The string whose method is called is inserted in between each given string.
|      The result is returned as a new string.
|
|      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
|
|  ljust(self, width, fillchar=' ', /)
|      Return a left-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  lower(self, /)
|      Return a copy of the string converted to lowercase.
|
|  lstrip(self, chars=None, /)
|      Return a copy of the string with leading whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  partition(self, sep, /)
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string.  If the separator is found,
|      returns a 3-tuple containing the part before the separator, the separator
|      itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing the original string
|      and two empty strings.
```

```
|
|  removeprefix(self, prefix, /)
|      Return a str with the given prefix string removed if present.
|
|      If the string starts with the prefix string, return string[len(prefix):].
|      Otherwise, return a copy of the original string.
|
|  removesuffix(self, suffix, /)
|      Return a str with the given suffix string removed if present.
|
|      If the string ends with the suffix string and that suffix is not empty,
|      return string[:-len(suffix)]. Otherwise, return a copy of the original
|      string.
|
|  replace(self, old, new, count=-1, /)
|      Return a copy with all occurrences of substring old replaced by new.
|
|        count
|          Maximum number of occurrences to replace.
|          -1 (the default value) means replace all occurrences.
|
|      If the optional argument count is given, only the first count occurrences are
|      replaced.
|
|  rfind(...)
|      S.rfind(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  rindex(...)
|      S.rindex(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  rjust(self, width, fillchar=' ', /)
|      Return a right-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
```

```
|
|  rpartition(self, sep, /)
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string, starting at the end. If
|      the separator is found, returns a 3-tuple containing the part before the
|      separator, the separator itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing two empty strings
|      and the original string.
|
|  rsplit(self, /, sep=None, maxsplit=-1)
|      Return a list of the substrings in the string, using sep as the separator string.
|
|        sep
|          The separator used to split the string.
|
|          When set to None (the default value), will split on any whitespace
|          character (including \\n \\r \\t \\f and spaces) and will discard
|          empty strings from the result.
|        maxsplit
|          Maximum number of splits (starting from the left).
|          -1 (the default value) means no limit.
|
|      Splitting starts at the end of the string and works to the front.
|
|  rstrip(self, chars=None, /)
|      Return a copy of the string with trailing whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  split(self, /, sep=None, maxsplit=-1)
|      Return a list of the substrings in the string, using sep as the separator string.
|
|        sep
|          The separator used to split the string.
|
|          When set to None (the default value), will split on any whitespace
|          character (including \\n \\r \\t \\f and spaces) and will discard
|          empty strings from the result.
|        maxsplit
|          Maximum number of splits (starting from the left).
|          -1 (the default value) means no limit.
|
|      Note, str.split() is mainly useful for data that has been intentionally
|      delimited.  With natural text that includes punctuation, consider using
```

```
|      the regular expression module.
|
|  splitlines(self, /, keepends=False)
|      Return a list of the lines in the string, breaking at line boundaries.
|
|      Line breaks are not included in the resulting list unless keepends is given and
|      true.
|
|  startswith(...)
|      S.startswith(prefix[, start[, end]]) -> bool
|
|      Return True if S starts with the specified prefix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      prefix can also be a tuple of strings to try.
|
|  strip(self, chars=None, /)
|      Return a copy of the string with leading and trailing whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  swapcase(self, /)
|      Convert uppercase characters to lowercase and lowercase characters to uppercase.
|
|  title(self, /)
|      Return a version of the string where each word is titlecased.
|
|      More specifically, words start with uppercased characters and all remaining
|      cased characters have lower case.
|
|  translate(self, table, /)
|      Replace each character in the string using the given translation table.
|
|        table
|          Translation table, which must be a mapping of Unicode ordinals to
|          Unicode ordinals, strings, or None.
|
|      The table must implement lookup/indexing via __getitem__, for instance a
|      dictionary or list.  If this operation raises LookupError, the character is
|      left untouched.  Characters mapped to None are deleted.
|
|  upper(self, /)
|      Return a copy of the string converted to uppercase.
|
|  zfill(self, width, /)
|      Pad a numeric string with zeros on the left, to fill a field of the given width.
```

```
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  maketrans(...)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary mapping Unicode
 |      ordinals (integers) or characters to Unicode ordinals, strings or None.
 |      Character keys will be then converted to ordinals.
 |      If there are two arguments, they must be strings of equal length, and
 |      in the resulting dictionary, each character in x will be mapped to the
 |      character at the same position in y. If there is a third argument, it
 |      must be a string, whose characters will be mapped to None in the result.
```

```python
# Maak een nieuw objecttype aan met de naam Sample
class Sample:
    pass

# Een instantie van de klas Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

Volgens conventies geven we klassen een naam die begint met een **hoofdletter**. Merk op hoe x nu de referentie is naar onze nieuwe instantie van een voorbeeldklasse. Met andere woorden, we **instantiëren** de Sample-klasse.

Binnen in de klas hebben we momenteel alleen pas. Maar we kunnen klassenattributen en -methoden definiëren.

Een **attribuut** is een kenmerk (characteristic) van een object.

Een **methode** is een bewerking/operatie die we met het object kunnen uitvoeren.

We kunnen bijvoorbeeld een klasse maken met de naam Dog. Een kenmerk van een hond kan zijn ras of zijn naam zijn, terwijl een methode van een hond kan worden gedefinieerd door een .bark() -methode die een geluid retourneert.

Laten we aan de hand van een voorbeeld een beter begrip van attributen krijgen.

Attributen

De syntaxis voor het maken van een attribuut is:

```
self.attribute = something
```

Er is een speciale methode genaamd:

```
__init__()
```

Deze methode wordt gebruikt om de attributen van een object te **initialiseren**. Bijvoorbeeld:

```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Laten we opsplitsen wat we hierboven hebben. De speciale methode:

```
__init__()
```

wordt automatisch aangeroepen direct nadat het object is gemaakt:

```
def __init__(self, breed):
```

Elk attribuut in een klassendefinitie begint met een verwijzing naar het instantieobject. Het wordt volgens afspraak zelf genoemd. Het ras is het argument. De waarde wordt doorgegeven tijdens de instantie van de klasse.

```
self.breed = breed
```

Nu hebben we twee instanties van de klasse Dog gemaakt. Met twee rassen (breed) hebben we dan als volgt toegang tot deze attributen:

```
sam.breed
```

```
'Lab'
```

```
frank.breed
```

```
'Huskie'
```

Merk op dat we geen haakjes hebben na 'breed'; dit komt omdat het een **attribuut** is en er **geen argumenten** voor nodig zijn.

In Python zijn er ook *Class Object Attributes*. Deze klassenobjectkenmerken zijn hetzelfde voor elk exemplaar van de klasse. We kunnen bijvoorbeeld het attribuut *species* maken voor de klasse 'Dog'. Honden, ongeacht hun ras, naam of andere eigenschappen, zullen altijd zoogdieren (mammals) zijn. We passen deze logica op de volgende manier toe:

```
class Dog:

    # Class Object Attribute
    species = 'mammal'

    def __init__(self,breed,name):
        self.breed = breed
        self.name = name
```

```
sam = Dog('Lab','Sam')
```

```
sam.name
```

```
'Sam'
```

Merk op dat het class-object-attribuut buiten alle methoden in de klasse wordt gedefinieerd. Ook volgens conventies plaatsen we ze eerst voor de init.

```
sam.species
```

```
'mammal'
```

Methoden

Methoden zijn **functies** die zijn gedefinieerd in de **body** van een klasse. Ze worden gebruikt om operaties uit te voeren met de attributen van onze objecten. Methoden zijn een sleutelconcept van het OOP-paradigma. Ze zijn essentieel voor het verdelen van verantwoordelijkheden bij het programmeren, vooral bij grote toepassingen.

U kunt methoden zien als functies die op een object inwerken (acting) en waarbij rekening wordt gehouden met het object zelf via het *self*-argument.

Laten we een voorbeeld bekijken van het maken van een Circle-klasse:

```
class Circle:
    pi = 3.14

    # Cirkel wordt geïnstantieerd met een straal (radius) (standaard is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Methode voor het resetten van Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Methode voor het verkrijgen van omtrek (circumference)
    def getCircumference(self):
```

```
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In de ___init___ methode hierboven moesten we Circle.pi aanroepen om het oppervlakteattribuut (area) te berekenen. Dit komt omdat het object nog geen eigen .pi-attribuut heeft, dus we noemen het Class Object Attribute pi in plaats daarvan. In de setRadius-methode werken we echter met een bestaand Circle-object dat wel zijn eigen pi-attribuut heeft. Hier kunnen we Circle.pi of self.pi gebruiken. Laten we nu de straal (radius) veranderen en kijken hoe dat ons Circle-object beïnvloedt (affect):

```
c.setRadius(2)

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Geweldig! Merk op hoe we hiervoor 'self'. notatie gebruikten om te verwijzen naar attributen van de klasse binnen de methode-aanroepen. Bekijk hoe de bovenstaande code werkt en probeer uw eigen methode te maken.

Inheritance (Overerving)

Overerving is een manier om nieuwe klassen te vormen met behulp van klassen die al zijn gedefinieerd. De nieuw gevormde klassen worden **afgeleide** (derived-class) klassen genoemd, de klassen waaruit we afleiden worden **basisklassen** (base-class) genoemd. Belangrijke voordelen van overerving zijn **codehergebruik** en **vermindering van de complexiteit** van een programma. De afgeleide klassen (descendants) overschrijven of breiden de functionaliteit van basisklassen (ancestors) uit.

Laten we een voorbeeld bekijken door ons eerdere werk over de Dog-klasse op te nemen:

```
class Animal:
```

```
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

```
d = Dog()
```

```
Animal created
Dog created
```

```
d.whoAmI()
```

```
Dog
```

```
d.eat()
```

```
Eating
```

```
d.bark()
```

```
Woof!
```

In dit voorbeeld hebben we 2 klassen: Animal en Dog. Animal is de basisklasse (base), Dog is de afgeleide (derived) klasse.

De afgeleide klasse erft de functionaliteit van de basisklasse.

- Het wordt getoond door de eat() methode.

De afgeleide klasse wijzigt het bestaande gedrag (behaviour) van de basisklasse.

- getoond door de whoAmI() methode.

Ten slotte breidt de afgeleide klasse de functionaliteit van de basisklasse uit door een nieuwe bark()-methode te definiëren.

Polymorfisme

We hebben geleerd dat hoewel functies verschillende argumenten kunnen bevatten, methoden behoren tot de objecten waarop ze werken. In Python verwijst *polymorfisme* naar de manier waarop verschillende objectklassen dezelfde methodenaam kunnen delen, en die methoden kunnen vanaf dezelfde plaats worden aangeroepen, ook al kunnen er verschillende objecten worden ingevoerd. De beste manier om dit uit te leggen is bij voorbeeld:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Hier hebben we een Dog-klasse en een Cat-klasse, en elk heeft een `.speak()`-methode. Wanneer **aangeroepen**, retourneert de `.speak()`-methode van elk object een resultaat dat **uniek** is voor het object.

Er zijn een paar verschillende manieren om polymorfisme aan te tonen. Eerst met een for-lus:

```
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Een andere is met functies:

```
def pet_speak(pet):
    print(pet.speak())
```

```
pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In beide gevallen konden we verschillende objecttypes doorgeven en kregen we objectspecifieke resultaten van hetzelfde mechanisme.

Een meer gebruikelijke (common) praktijk is om abstracte klassen en overerving te gebruiken. Een abstracte klasse is er een die **nooit verwacht** te worden **geïnstantieerd**. We zullen bijvoorbeeld nooit een Animal-object hebben, alleen Dog- en Cat-objecten, hoewel Dogs and Cats zijn afgeleid van Animals:

```
class Animal:
    def __init__(self, name):    # Constructeur van de klas
        self.name = name

    def speak(self):             # Abstracte methode, alleen gedefinieerd door conventie
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Voorbeelden van polymorfisme uit het levensecht-applicaties zijn als de volgende: * Voor het openen van de verschillende bestandstypen: Er zijn verschillende tools nodig om Word-, pdf- en Excel-bestanden weer te geven * Voor het toevoeging van de verschillende objecten: De operator + voert rekenkundige bewerkingen (arithmetic) en aaneenschakelingen (concatenation) uit

Speciale methoden met ___ methode-naam ___

Laten we tot slot speciale methoden bespreken. Klassen in Python kunnen bepaalde bewerkingen uitvoeren met speciale methodenamen. Deze methoden worden niet rechtstreeks aangeroepen, maar door Python-specifieke taal-syntaxis. Laten we bijvoorbeeld een Boek-klas maken:

```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author, self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")

book = Book("Python Rocks!", "Jose Portilla", 159)

# Speciale methoden
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

De methoden **init**(), **str**(), **len**() en **del**()

Deze speciale methoden worden gedefinieerd door het gebruik van **onderstrepingstekens** (underscore). Ze stellen ons in staat om Python-specifieke functies te gebruiken op objecten die met onze klasse zijn gemaakt.

Geweldig! Na deze lezing zou u een basiskennis moeten hebben van hoe u uw eigen objecten kunt maken met klasse in Python. U zult hier intensief gebruik van maken in uw volgende Milestone-Project!

Ga voor meer geweldige bronnen (resources) over dit onderwerp naar:

Mozilla's bericht

Zelfstudie-portaal

Officiële documentatie