In [1]:
```
# Scenario:
# there are several optional parameters and
# we want to provide a value for one of the later parameters
# while not providing a value for the earlier ones
# Solution: use keyword arguments (named arguments) in function call

def fn(x, y=55, z=66, t=77):
    print(f"{x:>5}{y:>5}{z:>5}{t:>5}")

print(f"{'x':>5}{'y':>5}{'z':>5}{'t':>5}")

fn(5)           # 1 positional argument

fn(x=6)         # 1 keyword argument

fn(x=7, z=8)    # 2 keyword arguments

fn(z=8, x=7)    # 2 keyword arguments

fn(10, 20, 30)    # 3 positional arguments

fn(40, y=50)    # 1 positional, 1 keyword

# fn()          # required argument missing

# fn(x=5,6)     # non-keyword argument after a keyword argument

# fn(10, x=20)  # duplicate value for the same argument

# f(a=10)      # unknown keyword argument
```

```
    x    y    z    t
    5   55   66   77
    6   55   66   77
    7   55    8   77
    7   55    8   77
   10   20   30   77
   40   50   66   77
```

In [2]:
```
# In a function call,
# keyword arguments must follow positional arguments
# all keyword arguments must match one of the parameters accepted by the function
# order is not important
# no argument may receive a value more than once
```

In [3]:
```
# Scenario: we want to capture an argument that
# we might not know in advance (i.e, datatype/length)?

# arbitrary argument lists - variadic parameters

# when a final formal parameter of the form **name is present,
# it receives a dictionary containing all keyword arguments
# except for those corresponding to a formal parameter

# a formal parameter of the form *name receives a tuple
# containing the positional arguments beyond the formal parameter list

# *name must occur before **name
```

In [4]:
```
def f(x, *args, **kwargs):
    print(f"x: {x}")
```

```
    for arg in args:
        print(arg)
    print("-" * 10)

    for kw in kwargs:
        print(f"{kw}: {kwargs[kw]}")

f(5,6,7,8,a=10,b=20)
```

```
x: 5
6
7
8
----------
a: 10
b: 20
```

In [5]:
```
def f(x, *args):
    print(f"x: {x}")

    for arg in args:
        print(arg)

f(5,6,7,8)
```

```
x: 5
6
7
8
```

In [6]:
```
def f(x, **kwargs):
    print(f"x: {x}")

    for kw in kwargs:
        print(f"{kw}: {kwargs[kw]}")

f(5,a=10,b=20,c=30)
```

```
x: 5
a: 10
b: 20
c: 30
```

In [7]:
```
# unpacking argument lists
# Scenario: arguments are already in a list or tuple
# but need to be unpacked for a function call
# requiring separate positional arguments

args = (2, 20, 3)
list(range(*args))            # call with arguments unpacked from a list
```

Out[7]:
```
[2, 5, 8, 11, 14, 17]
```

In [8]:
```
# unpacking argument lists
# similarly, dictionaries can deliver keyword arguments with the ** operator

def f(roll, name, age):
    print(roll,name, age)

d = {'roll': 5, 'name': 'Aditi', 'age': 22}
f(**d)
```

```
5 Aditi 22
```

In [ ]:
```
'''
By default, arguments may be passed to a Python function
either by position or explicitly by keyword.
For readability and performance, it makes sense to restrict
the way arguments can be passed so that a developer
need only look at the function definition to determine
if items are passed by position, by position or keyword, or by keyword.
'''
```

In [ ]:
```
# Special parameters

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only

# / and * are optional.
# if / and * are not present in the function definition,
# arguments may be passed to a function by position or by keyword.
# Keyword parameters are also referred to as named parameters.
```

In [9]:
```
# Positional-Only Parameters
# parameters' order matters
# parameters cannot be passed by keyword
# placed before a /

# Keyword-Only
# an * in the arguments list just before the first keyword-only parameter
```

In [10]:
```
def standard_arg(arg):
    print(arg)

standard_arg(5)
standard_arg(arg=6)
```

```
5
6
```

In [11]:
```
def pos_only_arg(arg, /):
    print(arg)

pos_only_arg(5)
# pos_only_arg(arg=5)   # TypeError: pos_only_arg() got some
                        # positional-only arguments passed as
                        # keyword arguments: 'arg'
```

```
5
```

In [12]:
```
def kwd_only_arg(*, arg):
    print(arg)

# kwd_only_arg(5)   # TypeError: kwd_only_arg() takes 0 positional
                    # arguments but 1 was given
kwd_only_arg(arg=5)
```

```
5
```

In [13]:
```
def combined_example(pos_only, /, standard, *, kwd_only):
    print(f"{pos_only} {standard} {kwd_only}")

# combined_example(1, 2, 3)     #TypeError: combined_example()
```

```
                                # takes 2 positional arguments but 3 were given

combined_example(3, 4, kwd_only=5)

combined_example(6, standard=7, kwd_only=8)

# combined_example(pos_only=1, standard=2, kwd_only=3)
# TypeError: combined_example() got some positional-only arguments
# passed as keyword arguments: 'pos_only'
```

```
3 4 5
6 7 8
```

In [14]:
```
# potential collision between the positional argument 'name'
# and **kwds which has 'name' as a key

def f(name, **kwds):
    return 'name' in kwds

# f(1, **{'name': 2}) # TypeError: f() got multiple values for argument 'name'
```

In [15]:
```
# Workaround
# using / (positional only arguments),
# possible - it allows name as a positional argument and 'name'
# as a key in the keyword arguments

def f(name, /, **kwds):
    return 'name' in kwds

f(1, **{'name': 2})
```

Out[15]:
```
True
```

In [16]:
```
# Which to use

# positional-only
# if name of the parameters need not be available to the user.
# (useful when parameter names have no real meaning),
# to enforce the order of the arguments when the function is called
# if needed to take some positional parameters and arbitrary keywords.

# keyword-only
# when names have meaning and
# the function definition is more understandable by being explicit with names
# prevent users from relying on the position of the argument being passed.

# For an API, use positional-only to prevent breaking API changes if the
# parameter's name is modified in the future.
```

In [17]:
```
# lambda expressions
# anonymous functions can be created with the lambda keyword
# lambda arguments: return value
# lambda functions can be used wherever function objects are required
# With a lambda function, we can execute the function immediately after
# its creation and receive the result;
# Immediately Invoked Function Execution (IIFE).

print(lambda x: x-2)
print(type(lambda x: x-2))
print((lambda x: x-2)(6))
```

```
<function <lambda> at 0x000002979C0C5E10>
<class 'function'>
4
```

In [18]:
```python
# Can assign a lambda function to a variable and
# then call that variable as a normal function

squared = lambda x: x**2
squared(5)

# But it is considered a bad practice according to
# PEP 8 - Style guide for Python Code (https://peps.python.org/pep-0008/)
# "The use of the assignment statement eliminates the sole benefit
# a lambda expression can offer over an
# explicit def statement (i.e. that it can be embedded inside a larger expression)"

# PEP stands for Python Enhancement Proposal.
# A PEP is a design document providing information to the Python community,
# or describing a new feature for Python or its processes or environment.
```

Out[18]:
```
25
```

In [19]:
```python
# So, according to PEP 8,

# Correct:
def squared(x): return x**2

# Wrong:
squared = lambda x: x**2

# The first form means that the name of the resulting function object
# is specifically 'squared' instead of the generic '<lambda>'.
# This is more useful for tracebacks and string representations in general.
```

In [20]:
```python
# lambda expression to return a function

def powered(n):
    return lambda x: x ** n

f = powered(3)

print(f(2))
print(f(5))
```

```
8
125
```